

UNCLASSIFIED

AD NUMBER

AD825963

LIMITATION CHANGES

TO:

Approved for public release; distribution is unlimited.

FROM:

Distribution authorized to U.S. Gov't. agencies and their contractors;  
Administrative/Operational Use; NOV 1967. Other requests shall be referred to Office of the Chief of Research and Development (Army), Washington, DC 20310.

AUTHORITY

AROD ltr 4 Aug 1971

THIS PAGE IS UNCLASSIFIED

ARO-D Report 67-3

AD825963

**PROCEEDINGS OF THE 1967 ARMY NUMERICAL  
ANALYSIS CONFERENCE**



This document is subject to special export controls and each transmittal to foreign governments or foreign nationals may be made only with prior approval of the U. S. Army Research Office—Durham, Durham, North Carolina.

The findings in this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.

**Sponsored by  
The Army Mathematics Steering Committee  
on Behalf of**

**THE OFFICE OF THE CHIEF OF RESEARCH AND DEVELOPMENT**

JAN 31 1968

U. S. ARMY RESEARCH OFFICE-DURHAM

Report No. 67-3

November 1967

PROCEEDINGS OF THE 1967 ARMY NUMERICAL  
ANALYSIS CONFERENCE

Sponsored by the Army Mathematics Steering Committee

Host

U.S. Army Mathematics Research Center  
University of Wisconsin, Madison, Wisconsin

25-26 May 1967

This document is subject to special export controls and each transmittal to foreign governments or foreign nationals may be made only with prior approval of the U. S. Army Research Office—Durham, Durham, North Carolina.

The findings in this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.

U. S. Army Research Office-Durham  
Box CM, Duke Station  
Durham, North Carolina

## FOREWORD

Several years ago the Office of Ordnance Research (now the Army Research Office-Durham) organized an OOR Liaison Group on Computers. Two meetings of this group were held, one in 1959 and the other in 1960, to exchange information of interest to managers of ordnance "other than business" computers. The Army Mathematics Steering Committee asked that these meetings be revived and placed on an army-wide basis. The first two meetings in this new series were held, one in 1962 at ARO-D and the other in 1964 at the Harry Diamond Laboratories and the National Bureau of Standards, under the title "ARO Working Group on Computers". The 1965 Conference was held at the Ballistic Research Laboratories under the present title of this series; namely, "Army Numerical Analysis Conference". The 1966 meeting was conducted at the U. S. Army Research Personnel Office, Washington, D. C.

The U. S. Army Mathematics Research Center, University of Wisconsin, served as the host for the 1967 Army Numerical Analysis Conference. It was held at the Wisconsin Center on 25-26 May 1967, and was attended by over 58 scientists. The three invited addresses were delivered by Professor W. Kahan, D. F. Kennedy, and Dr. Allen Reiter. They treated respectively topics on numerical solutions of polynomial equations, COSMIC, and interval arithmetic. Besides these talks there were nine contributed papers.

Dr. Louis B. Rall served as Chairman on Local Arrangements. Those in attendance were indebted to him, not only for excellent accommodations at the meeting, but also for organizing a large portion of the program.

The Chairman of the conference, Dr. John H. Giese, has asked that the proceedings of this meeting be published and issued to interested army scientists. He would like to thank, on behalf of the Army Mathematics Steering Committee, the sponsor of these conferences, all the speakers for their very interesting papers and the various chairmen for their help in conducting this meeting. Thanks are also due to Professor S. C. Kleene, Acting Director of MRC, for his interesting welcoming remarks and for having his installation serve as host for this conference.



## TABLE OF CONTENTS

Title	Page
Foreword.....	i
Table of Contents.....	iii
Program.....	v
COSMIC: A Center for the Dissemination of Computer Programs and Computer Information	
by D.F. Kennedy.....	1
Machine Language Programming - How and Why	
by J.M. Yohe.....	3
NEWTON: A General Purpose Program for Solving Nonlinear Systems	
by J.H. Gray and L.B. Rall.....	11
Experience with FORMAC at HDL	
by D.S. Marsh.....	61
A Simple Electronic True Random Event Generator	
by R.G. Polk, D.R. Koehler, and J.T. Grissom.....	75
Programming Interval Arithmetic and Applications	
by Allen Reiter.....	87
Homeostatic Organizations of Adaptive Parallel Processing Systems	
by R.M. Dunn.....	99
Logical Structure of an Automatically Sequenced Explosive Control Device	
by S.H. Eisman.....	111
Problem Solving Digital-Analogue Simulation	
by H.W. Bloom*.....	141
A Shell Computer Program which Determines the Physical Properties of an Artillery Shell and Represents Its Dimensions Graphically	
by Forrest McMains.....	143
Zerpol, A Zero Finding Algorithm for Polynomials Using Laguerre's Method	
by B.T. Smith.....	153

---

\* This paper was presented at the Conference. It does not appear in the Proceedings.

The Numerical Solution of Polynomial Equations by W. Kahan.....	175
Roundoff Errors by Ben Noble.....	209
Attendance List.....	229

## PROGRAM

Wednesday, 24 May 1967

2000-2200 Open House, Fourth Floor Lounge, Mathematics Research Center

(All sessions held in the Auditorium of the Wisconsin Center, Lake and Langdon Streets)

Thursday, 25 May 1967

0830-0900 Registration: First Floor, Wisconsin Center, Exhibit Gallery

### SESSION I

Chairman: Dr. John H. Giese, Ballistics Research Lab.,  
Aberdeen Proving Ground, Maryland

0900-0915 Welcoming Remarks

0915-1000 COSMIC: A Center for the Dissemination of Computer  
Programs and Computer Information  
Mr. D. F. Kennedy, University of Georgia Computing  
Center, Athens, Georgia

1000-1030 Coffee Break - Exhibit Gallery

1030-1100 Machine Language Programming - How and Why  
J. M. Yohe, Mathematics Research Center

1100-1130 NEWTON: A General Purpose Program for Solving Nonlinear  
Systems  
Julia H. Gray and L. B. Rall, Mathematics Research Center

1130-1215 Experience with FORMAC at HDL  
D. S. Marsh, Harry Diamond Laboratories, Washington, DC

1215-1330 Lunch

### SESSION II

Chairman: Dr. Donald Greenspan, Mathematics Research  
Center

- 1330-1400 A Simple Electronic True Random Event Generator  
R. G. Polk, D. R. Koehler, and J. T. Grissom,  
Redstone Arsenal, Alabama
- 1400-1500 Programming Interval Arithmetic and Applications  
Dr. Allen Reiter, Lockheed Missile and Space Company,  
Palo Alto, California
- 1500-1530 Coffee Break
- 1530-1610 Homeostatic Organizations of Adaptive Parallel Processing  
Systems  
Robert M. Dunn, U. S. Army Electronics Command,  
Fort Monmouth, New Jersey
- 1610-1700 Logical Structure of an Automatically Sequenced Explosive  
Control Device  
S. H. Eisman, Frankford Arsenal, Philadelphia,  
Pennsylvania

Friday, 26 May 1967

SESSION III

- Chairman: Dr. Roger A. MacGowan, Army Missile Command,  
Redstone Arsenal, Alabama
- 0830-0920 Problem Solving Digital-Analogue Simulation  
Howard W. Bloom, Harry Diamond Laboratories,  
Washington, D. C.
- 0920-1000 A Shell Computer Program which Determines the Physical  
Properties of an Artillery Shell and Represents Its Dimen-  
sions Graphically  
Forrest McMains, Picatinny Arsenal, Dover, New Jersey
- 1000-1030 Coffee Break
- 1030-1130 The Numerical Solution of Polynomial Equations  
Professor W. Kahan, University of Toronto, Toronto,  
Canada
- 1130-1230 Roundoff Errors  
Professor Ben Noble, Mathematics Research Center and  
the University of Wisconsin, Madison, Wisconsin

COMPUTER SOFTWARE MANAGEMENT AND INFORMATION CENTER  
COSMIC

Donald F. Kennedy  
COSMIC, The University of Georgia Computer Center  
Athens, Georgia

INTRODUCTION. In July 1966, The University of Georgia was awarded a contract by the National Aeronautics and Space Administration to establish and operate a center for the dissemination of computer programs and computer information. This center, known as Computer Software Management and Information Center (COSMIC), is working through the NASA Technology Utilization Office at the Marshall Space Flight Center in conjunction with other NASA Centers and NASA Headquarters. Through this joint effort, computer programs and computer information developed by or for NASA are made available, at minimal costs, to potential users in industry, business, education, and other sectors of our economy. In addition, computer programs developed by or for the Atomic Energy Commission, which is participating in the NASA Technology Utilization Program, are also made available through COSMIC.

PURPOSE. One of the primary functions of the NASA Technology Utilization Program is to identify technological advances derived from the space effort and to make them available for use by industry and business. One of the most useful sources of technical aid and information to many organizations is a wide range of well documented, operational computer programs and computer information. By making these computer programs, which are classified as new technology, available to industry and business, NASA hopes to contribute directly to the national industrial effort and offer companies the opportunity to avoid duplication and to shorten the task of developing computer programs.

EXPERIENCE. The Computer Center at the University of Georgia has had extensive experience over the past four years in providing computer services and assistance in computer applications to approximately sixty industrial and business firms. The Center employs a professional staff of statisticians, mathematicians, biologists, numerical analysts, engineers, chemists, physicists, and information and computer scientists. The two major computer systems in the Center are the IBM 360 Model 65 and the IBM 7094 with two IBM 1401 systems serving as input/output peripheral units for the 7094. In addition, an IBM 1620 computer and an EAI TR-20 analog computer are operated on an open-shop basis.

PROCEDURE. Under the original contract, NASA performed the evaluation of computer programs and forwarded to COSMIC only those programs and documentations which were to be included in the COSMIC library. However, under a modification of the contract in December 1966, the University of Georgia was given the additional responsibility of evaluating NASA computer programs. Documentation on each program is forwarded to COSMIC for evaluation to determine its applicability to a variety of uses for industry, business, and education. If the program is found applicable, a more in-depth evaluation is performed considering such factors as soundness of logic, accuracy of output, and completeness of the documentation. After the evaluation, a recommendation is made to NASA as to the inclusion or

rejection of the program. If the computer program is included in the COSMIC library, it is announced to industry and business by both NASA and COSMIC.

In addition to the NASA computer programs, COSMIC has it in its library computer programs obtained from business and educational firms.

The programs are disseminated on tape or in card form, depending upon the requestor's preference. Each requestor is charged for the reproduction, handling, and mailing of programs.

Documentation may be requested without a program, if desired. Originally, documentation was disseminated at no charge to the requestor; however, in an effort to become self-supporting, COSMIC has instituted a sliding-scale charge for documentation based on a fee of 6 cents per page.

A directory of abstracts of computer programs available from COSMIC is disseminated periodically. Interested parties can receive a complimentary copy by writing to:

COSMIC,  
Computer Center,  
University of Georgia  
Athens, Georgia 30601

CONCLUSION. During the first twelve months of operation, COSMIC has had great success and growth. It has received requests for programs and information from every section of the nation and, in fact, from every part of the world, even from countries behind the iron curtain.

Based on its present success and growth, COSMIC should become the largest disseminator of computer programs and computer information in the nation and should have one of the most complete libraries of computer programs in the nation.

## MACHINE LANGUAGE PROGRAMMING HOW AND WHY\*

J. M. Yohe  
Mathematics Research Center, U. S. Army  
Madison, Wisconsin

There seems to be a feeling in some quarters that Machine Language programming is obsolete -- or at least, that it is no longer useful for everyday applications. This feeling is largely due to the availability of powerful problem-oriented languages such as FORTRAN, COBOL, ALGOL, and others. With these languages in common use, the argument goes, a person needs no knowledge of Machine Language; the compiler does all of the "dirty work".

This is evidenced by the increasing difficulty of using machine language in programming. For example, when the CDC 1604 computer was first installed here at the University of Wisconsin, the FORTRAN compiler allowed a programmer to intermix machine language and FORTRAN statements. However, when an improved FORTRAN compiler was released, this capability was missing. And in some installations, the use of machine language programming is actively discouraged.

It is indeed tempting to believe that machine language programming is obsolete, as anyone who has ever done any machine language programming will attest. There is a considerable amount of boring detail connected with writing a program in machine language, and I am the first to want to dispense with it. However, I don't believe that machine language is dead yet, nor do I believe that the need for it will disappear in the near future. I feel that every programmer should know something about programming in machine language, even if he never uses it. And I believe that, in most cases, significant savings in computer time, man-hours, and dollars can result from judicious use of machine language. There are two major reasons for this contention: First, a programmer who knows machine language can write more efficient programs than one who does not know machine language. This is true whether he writes his programs in machine language or in one of the problem oriented languages. Second, a knowledge of machine language can be of great help in debugging programs, whether they are written in machine language or not.

There are still other benefits to be derived from machine language programming, as we shall see presently.

---

\*Sponsored by the Mathematics Research Center, U. S. Army, Madison, Wisconsin under Contract No.: DA-31-124-ARO-D-462.



I do not intend to take anything away from those who conceive, implement, and use the problem oriented languages. On the contrary, I feel that these languages are vital. I would even go so far as to say that perhaps most computer programming should be done in these languages. I do want to convince you that these languages are not yet the answer to all programming problems.

Let us first make a few remarks about how a person can go about acquiring a knowledge of machine language programming.

Perhaps the most important comment is that machine language programming, like any other discipline, cannot be taught -- a person must learn it. In learning, motivation is an important factor; the best motivation for learning machine language programming is a need to know it. So if you supervisors want machine language programming to be used in your installation, I urge you to encourage your programmers to use it in those situations where it would be of value.

The best way to learn machine language programming is from an experienced programmer in a working situation. The person who is writing a machine language program and has access to an experienced programmer will learn programming quite rapidly. Barring that, some textbooks can give a person a good grounding in the fundamentals of machine language programming, and for certain computers, there are handbooks available for learning -- for example, Machine Language Programming for the CDC 3600, MRC Technical Summary Report No. 721, which will appear shortly. The computer reference manual is usually one of the least effective ways of learning machine language, but it will do in the absence of any other source.

The only really effective way of learning machine language programming, however, is by doing it.

Why is machine language programming worth consideration?

There are several reasons. First and perhaps foremost, machine language programs can be considerably more efficient than even the most skillfully written programs in problem oriented languages. The compilers, after all, are general purpose programs, designed to handle a wide variety of cases with acceptable efficiency. They cannot, therefore, tailor programs to specific situations; to do so would require additional logic in the compiler program to the point that the compiler would be cumbersome and quite slow. Consider, for example, the question of testing whether  $A = B$ . The usual method of making this test is to subtract B from A and test the result for zero, and this is quite an acceptable method. If, however, B happens to be



zero already, there is no need to do the subtraction; we need only test A for zero. However, many compilers do not even recognize this particularly simple special case; they will compile code to subtract zero from A and test the result for zero. Clearly, a person writing a program in machine language could easily eliminate the extra subtract instruction which the compiler would generate. Far greater economies are usually possible in more complex situations.

Another benefit derives from a programmer knowing machine language. A programmer who knows machine language can often write more efficient programs in a problem oriented language than a programmer who knows only the problem oriented language. The programmer familiar with machine language will know roughly how the compiler will translate the source statements he writes, and he will be able to avoid situations which cause unneeded instructions to be generated. He will understand, for example, exactly what is involved in mixed-mode arithmetic (for example, dividing a floating-point number by an integer) and will be able to make an educated decision about what course of action will result in the most efficient object program. Moreover, he will know when to use machine language and when to stick with the problem-oriented language.

A third and very important argument for a programmer's knowing machine language is that it will be of immeasurable value to him in debugging his programs, whether written in machine language or in a problem oriented language. He will be able, for example, to read core dumps, understand what kinds of errors might cause a certain wierd symptom, and even track down errors generated by library subroutines, the compiler or even the computer itself (in the rare instances when they occur).

We turn to a simple example. A program to clear an array to zero was written for the CDC 3600, first in FORTRAN using four different methods, and then in machine language. Let us examine the source statements and the code generated from them, and then the machine language code to do the same thing.

```
DO 10 I=1, 10000
10 A(I) = 0.0
```

#### Example 1

	ENA	1
	STA	=SI
	LIL	I, 1
	ENI	9999, 2
WS00001.	BSS	0
.10	ENA	0
	STA	A-1, 1
	INI	1, 1
	IJP	WS00001., 2

Example 1 is the traditional way of writing this program in FORTRAN, and, it turns out, is also the most efficient way of doing it in this compiler. Note, however, that the instruction ENA 0 (Enter A with zero) is executed on every pass through the loop, even though the A-register is never changed in the loop and thus always contains zero anyhow. Note also that two index registers are used, whereas one would have been sufficient.

```

      I=1
10  A(I) = 0.0
      I=I+1
      IF(I. LE. 10000) GO TO 10

```

#### Example 2

```

      ENA      1
      STA      =SI
.10  ENA      0
      LIL      I, 1
      STA      A-I, 1
      LDA      I
      INA      I
      STA      =SI
      LAC      I
      INA      10000
      AJP, ZR   .100001
      AJP, MI   .100002
.100001  SLJ      .10
.100002

```

In Example 2, the DO-loop logic was abandoned and indexing was done explicitly. This resulted in a far less efficient program, although a sophisticated compiler could have improved it considerably. For example, in this situation, the variable I could have been kept in an index register. Moreover, the variable I is already in the A-register when LAC I is executed; the compiler could have engineered matters so that I, rather than its complement, was used in the subsequent instructions, and thus eliminated the LAC I instruction. Note also that an extra jump instruction is executed at the end of the loop; AJP, ZR .100001 could equally well have read AJP, ZR .10-- or even been eliminated in this case.

```

      I=1
10 A(I) = 0.0
      I=I+1
      IF(I-10000) 10,10,20

```

### Example 3

```

      ENA      1
      STA      =SI
.10  ENA      0
      LIL      I,1
      STA      A-1,1
      LDA      I
      INA      I
      STA      =SI
      INA      -10000
      AJP,ZR   .10
      AJP,MI   .10

```

Example 3 differs from Example 2 only in the form of the IF statement. This form of the IF statement gave a more efficient object code, although many of the remarks concerning Example 2 apply equally well here.

```

      I=10000
10 A(I) = 0.0
      I=I-1
      IF(I.NE.0) GO TO 10

```

### Example 4

```

      ENA      10000
      STA      =SI
.10  ENA      0
      LIL      I,1
      STA      A-1,1
      LDA      I
      INA      -1
      STA      =SI
      INA      -0
      AJP,ZR   .100002
.100001  SLJ      .10
.100002

```

In Example 4, "reverse" indexing was used (as will be the case with Example 5, which is the machine language version of the program). Many of the remarks concerning Example 2 also apply to Example 4. Note here that the IF (I. NE. 0) statement generates an instruction which subtracts zero from I and then tests the result for zero. Note also that the construct

	AJP, ZR	. 100002
. 100001	SLJ	. 10

could have been replaced by the single instruction

AJP, NZ	. 10
---------	------

#### Example 5

	ENI	9999, 1
	ENA	0
L	STA	A, 1
	IJP	L, 1

Example 5 is the machine language version of the program. Observe that there are only two instructions in the loop, and that everything done in the loop must be done in the loop, while everything which can be done outside the loop is done outside the loop. This clearly results in a more efficient program than even the most efficient program generated by the FORTRAN compiler.

Comparing the most efficient FORTRAN program (Example 1) with the machine language program, (Example 5) we see that two extra instructions are executed on each pass through the loop. The execution time is about 2 $\mu$ s per pass. In 10,000 passes through the loop, this comes to about 30 milliseconds -- hardly worth considering. But if the procedure were to be executed a hundred thousand times, those two instructions would take 2,000 seconds on the 3600. At 11¢ per second, those two innocuous-looking instructions would cost \$220.00!

Let us now consider what types of programs should ordinarily be written in problem oriented languages and what types of programs stand to benefit from being written in machine language.

We first mention a few cases where machine language programming should not ordinarily be used. Programs which only need a couple of minutes of computer time can usually be written quite economically in one of the problem-oriented languages. The reason for this is that, in many of these cases, system overhead is responsible for a significant portion of the running time. There simply is not that much to be gained by speeding up the computation itself by a few seconds. Another case where machine language programming might be a mistake is when answers are needed in a hurry -- that is, when total turnaround time, rather than computer time, is the overriding consideration. In these cases, the longer time usually required to write and debug a machine language program might cause intolerable delays.

A third instance where machine language programming is not usually indicated is the case of the "one-shot" job, where the program will be abandoned or significantly changed after it has run successfully. In this case, the computer time necessary to debug a machine language program could well cancel any saving effected by writing the program in machine language.

Where, then, would machine language programming be worthwhile? The most obvious place is in programs which are to be used over a long period of time with no changes or only minor changes. If machine language programming can save 10% on a program which will run for, say, 1000 hours during a year's time, the total saving will be 100 hours. If the computer cost is \$200.00 per hour, this would result in a dollar saving of \$20,000.00. This is a realistic figure.

There are two other places where machine language programming can be of definite value. The first is the case where a problem can be handled far more efficiently by use of machine language programming than by the use of one of the problem oriented languages due to special circumstances. In this case, problems which were not economically feasible when programmed in one of the problem oriented languages can become quite reasonable when written in machine language. The second is the case where it is necessary to have complete control over the exact machine operations used as well as the sequence in which they are used. Such would be the case, for example, when the problem required strict control of round-off error. The program written here at MRC for Professor Lowell Schoenfeld to locate roots of the Riemann Zeta function falls into both of these categories.

Looking at the program for this Conference, the Newton program, to be described next, and Interval arithmetic, to be described this afternoon, both use machine language programming to good advantage; and in the analysis of round-off errors, which will be covered tomorrow, a knowledge of machine language for the computer in question is almost essential.

In summary, then, we have seen that knowledge of machine language can not only allow a programmer to write machine language programs when necessary, but it can also help him to write more efficient programs in any language, and help him debug programs more efficiently. This can result in significant savings in both time and money. This is why I claim that machine language programming is still very much alive.

**BLANK PAGE**

NEWTON: A GENERAL PURPOSE PROGRAM  
FOR SOLVING NONLINEAR SYSTEMS

Julia H. Gray and L. B. Rall

1. Introduction. A number of important problems which arise in practice may be reduced to the computational problem of solving a system of equations of the form

$$f_i(\xi_1, \xi_2, \dots, \xi_n) = 0, \quad i = 1, 2, \dots, n. \quad (1.1)$$

In (1.1), the functions  $f_i$  are assumed to be known, and the  $\xi_i$  are the unknowns,  $i = 1, 2, \dots, n$ . It may be supposed that all values are real, since in the case of complex values, (1.1) may be written as a system of  $2n$  real equations for the real and imaginary parts of the  $\xi_i$  by setting the real and imaginary parts of the  $f_i$  equal to zero,  $i = 1, 2, \dots, n$ .

This report describes an automatic computer program for solving systems of the form (1.1) which was developed at the Mathematics Research Center for the CDC 3600 computer operated by the University of Wisconsin Computing Center. The program is iterative in character; it starts from a given initial approximation and generates successive approximations to the solution of (1.1),

$$x^* = (\xi_1^*, \xi_2^*, \dots, \xi_n^*), \quad (1.2)$$

until pre-assigned criteria of accuracy are met, or until divergence is indicated.

---

Sponsored by the Mathematics Research Center, United States Army, Madison, Wisconsin, under Contract DA-31-124-ARO-D-462. This is MRC Technical Summary Report No. 790 July 1967.



In the latter case, the program prints an appropriate message. The convergence and error analyses are integral parts of the program. The program is general purpose in that it will handle any system of the form (1.1), up to the limits set by available core storage, in which the functions  $f_i$  can be written in terms of ordinary FORTRAN statements.

2. Theory. The system (1.1) can be considered to be an equation of the form

$$F(x) = 0 \quad (2.1)$$

in the space  $R^n$  of  $n$ -dimensional real vectors

$$x = (\xi_1, \xi_2, \dots, \xi_n) . \quad (2.2)$$

Here  $F$  is the vector function, or operator, defined by

$$F(x) = (f_1(x), f_2(x), \dots, f_n(x)) , \quad (2.3)$$

which maps the vector  $x$  into some other vector in  $R^n$ . A vector  $x$  will be a solution of equation (2.1) if it is mapped into the zero vector  $0 = (0, 0, \dots, 0)$  in  $R^n$  by the operator  $F$ .

The (Fréchet) derivative of the operator  $F$  is the  $n \times n$  matrix

$$F'(x) = \begin{pmatrix} \frac{\partial f_1}{\partial \xi_1} & \frac{\partial f_1}{\partial \xi_2} & \dots & \frac{\partial f_1}{\partial \xi_n} \\ \frac{\partial f_2}{\partial \xi_1} & \frac{\partial f_2}{\partial \xi_2} & \dots & \frac{\partial f_2}{\partial \xi_n} \\ \dots & \dots & \dots & \dots \\ \frac{\partial f_n}{\partial \xi_1} & \frac{\partial f_n}{\partial \xi_2} & \dots & \frac{\partial f_n}{\partial \xi_n} \end{pmatrix} , \quad (2.4)$$

or, for brevity,

$$F'(x) = \left( \frac{\partial f_i}{\partial \xi_j} \right) , \quad (2.5)$$



$i, j = 1, 2, \dots, n$  [1].  $F'(x)$  is sometimes called the Jacobian matrix of the system (1.1). The second derivative of  $F$  is the  $n \times n \times n$  array shown in Figure 1, or

$$F''(x) = \left( \frac{\partial^2 f_i}{\partial \xi_j \partial \xi_k} \right), \quad (2.6)$$

$i, j, k = 1, 2, \dots, n$  in condensed notation. For operational reasons, the convention

$$\frac{\partial^2 f_i}{\partial \xi_j \partial \xi_k} = \frac{\partial}{\partial \xi_k} \left( \frac{\partial f_i}{\partial \xi_j} \right) \quad (2.7)$$

is adopted.

The second derivative is a special type of bilinear operator

$$B = (b_{ijk}) \quad (2.8)$$

$i, j, k = 1, 2, \dots, n$  in  $R^n$  [1, 2].

In  $R^n$ , the norm  $\|x\|$  of a vector  $x$  will be defined to be

$$\|x\| = \max_{(i)} |\xi_i|. \quad (2.9)$$

Similarly, the norm  $\|A\|$  of an  $n \times n$  matrix  $A = (a_{ij})$  is taken to be

$$\|A\| = \max_{(i)} \sum_{j=1}^n |a_{ij}|, \quad (2.10)$$

and the norm  $\|B\|$  of an  $n \times n \times n$  bilinear operator  $B = (b_{ijk})$  is given by

$$\|B\| = \max_{(i)} \sum_{j=1}^n \sum_{k=1}^n |b_{ijk}|. \quad (2.11)$$

In (2.9) - (2.11), the index  $i$  runs over the integers  $1, 2, \dots, n$ .

If  $F$  is differentiable at

$$x_0 = (\xi_1^{(0)}, \xi_2^{(0)}, \dots, \xi_n^{(0)}), \quad (2.12)$$

$$F''(x) = \begin{bmatrix} \frac{\partial^2 f_1}{\partial \xi_1^2} & \frac{\partial^2 f_1}{\partial \xi_1 \partial \xi_2} & \dots & \frac{\partial^2 f_1}{\partial \xi_1 \partial \xi_n} & \left| \begin{array}{c} \frac{\partial^2 f_1}{\partial \xi_2 \partial \xi_1} & \dots & \frac{\partial^2 f_1}{\partial \xi_2 \partial \xi_n} \end{array} \right| & \dots & \frac{\partial^2 f_1}{\partial \xi_n^2} \\ \frac{\partial^2 f_2}{\partial \xi_1^2} & \frac{\partial^2 f_2}{\partial \xi_1 \partial \xi_2} & \dots & \frac{\partial^2 f_2}{\partial \xi_1 \partial \xi_n} & \left| \begin{array}{c} \frac{\partial^2 f_2}{\partial \xi_2 \partial \xi_1} & \dots & \frac{\partial^2 f_2}{\partial \xi_2 \partial \xi_n} \end{array} \right| & \dots & \frac{\partial^2 f_2}{\partial \xi_n^2} \\ \dots & \dots & \dots & \dots & \left| \begin{array}{c} \dots & \dots & \dots \end{array} \right| & \dots & \dots \\ \frac{\partial^2 f_n}{\partial \xi_1^2} & \frac{\partial^2 f_n}{\partial \xi_1 \partial \xi_2} & \dots & \frac{\partial^2 f_n}{\partial \xi_1 \partial \xi_n} & \left| \begin{array}{c} \frac{\partial^2 f_n}{\partial \xi_2 \partial \xi_1} & \dots & \frac{\partial^2 f_n}{\partial \xi_2 \partial \xi_n} \end{array} \right| & \dots & \frac{\partial^2 f_n}{\partial \xi_n^2} \end{bmatrix}$$

Figure I.  
The Second Derivative  $F''(x)$ .

then equation (2.1) may be written in the form

$$F(x_0) + F'(x_0)(x - x_0) + y_0 = 0 \quad (2.13)$$

in matrix-vector notation.  $F'(x_0)$  is obtained by evaluating the partial derivatives in (2.4) at  $x = x_0$ . The vector  $y_0$  is small relative to  $x - x_0$  in the sense that

$$\lim_{\|x - x_0\| \rightarrow 0} \frac{\|y_0\|}{\|x - x_0\|} = 0. \quad (2.14)$$

If a solution of equation (2.1) is close to  $x_0$ , one may feel justified in dropping  $y_0$  to obtain the approximate linear equation

$$F(x_0) + F'(x_0)(x - x_0) = 0, \quad (2.15)$$

the solution  $x = x_1$  of which will be

$$x_1 = x_0 - [F'(x_0)]^{-1} F(x_0), \quad (2.16)$$

provided that  $[F'(x_0)]^{-1}$  exists. Set

$$G_0 = (g_{ij}^{(0)}) = [F'(x_0)]^{-1}. \quad (2.17)$$

In terms of the original system (1.1),

$$x_1 = (\xi_1^{(1)}, \xi_2^{(1)}, \dots, \xi_n^{(1)}) \quad (2.18)$$

may be written

$$\xi_i^{(1)} = \xi_i^{(0)} - \sum_{j=1}^n g_{ij}^{(0)} f_j(\xi_1^{(0)}, \xi_2^{(0)}, \dots, \xi_n^{(0)}), \quad (2.19)$$

$$i = 1, 2, \dots, n.$$

On the assumption that  $x_1$  is a better approximation than  $x_0$  to a solution  $x = x^*$  of (2.1), the same process may be repeated with  $x_0$  replaced by  $x_1$  to obtain a further approximation  $x_2$ , and so on.

The generation of the sequence  $\{x_m\}$  of successive approximations by means of the relationship

$$x_{m+1} = x_m - [F'(x_m)]^{-1} F(x_m), \quad (2.20)$$

$m = 0, 1, 2, \dots$  is called Newton's method for solving equation (2.1). In order for the application of Newton's method to make sense from a computational standpoint, it is necessary to have affirmative answers to the following questions:

- (1) Does equation (2.1) have a solution  $x = x^*$  ?
- (2) Does the sequence generated by (2.20) exist and converge to  $x^*$  ?
- (3) Is it possible to obtain an estimate (that is, an upper bound) for the error  $\|x_m - x^*\|$  of approximation of  $x^*$  by  $x_m$ ,  $m = 0, 1, 2, \dots$  ?

At a given  $x_0$ , it is possible to settle these questions on the basis of a theorem due to L. V. Kantorovič [1, 2].

Theorem. At  $x = x_0$ , suppose that  $G_0 = [F'(x_0)]^{-1}$  exists,

$$\|G_0\| \leq B_0, \quad (2.21)$$

$$\|x_1 - x_0\| = \|-G_0 F(x_0)\| \leq \eta_0, \quad (2.22)$$

and

$$\|F''(x)\| \leq K \quad (2.23)$$

for  $x$  in the set

$$V(x_0, r) = \{x : \|x - x_0\| \leq r\}. \quad (2.24)$$

If

$$h_0 = B_0 \eta_0 K \leq \frac{1}{2} \quad (2.25)$$

and

$$r \geq r_0 = \frac{1 - \sqrt{1 - 2h_0}}{h_0} \eta_0, \quad (2.26)$$

Then:

- (1) Equation (2.21) has a solution  $x^*$  in  $V(x_0, r_0)$ ;
- (2) The Newton sequence  $\{x_m\}$  defined by (2.20) exists and converges to  $x^*$ ;
- (3) The error estimate

$$\|x^* - x_m\| \leq \frac{(2h_0)^{2^m - 1}}{2^{m-1}} \eta_0 \quad (2.27)$$

is valid, in particular,

$$\|x^* - x_1\| \leq 2h_0 \eta_0. \quad (2.28)$$

Proofs of this theorem may be found elsewhere [1, 2]. It is used as the basis for the optional automatic convergence and error analysis features of the computer program described in this report.

3. Generation of the Newton sequence. In order to generate the Newton sequence  $\{x_m\}$  defined by (2.20), subroutines are needed to perform the following operations:

- (1) Evaluate  $F(x_0)$ , that is, the  $n$  functions  $f_1(\xi_1^{(0)}, \xi_2^{(0)}, \dots, \xi_n^{(0)})$ ,  $i = 1, 2, \dots, n$ .
- (2) Evaluate  $F'(x_0)$ , which consists of the  $n^2$  functions

$$\frac{\partial f_i}{\partial \xi_j} = \frac{\partial f_i(\xi_1^{(0)}, \xi_2^{(0)}, \dots, \xi_n^{(0)})}{\partial \xi_j}, \quad (3.1)$$

$$i, j = 1, 2, \dots, n.$$

- (3) Invert the matrix  $F'(x_0)$ , if possible, and form the vector  $x_1$  defined by (2.16).

In addition, it is necessary to evaluate various norms in order to determine whether the iterative process should be continued with

$$x_0 := x_1 \quad (3.2)$$

or not.

In the operation of NEWTON, the user supplies the functions  $f_i$ ,  $i = 1, 2, \dots, n$ , written in a form suitable for compilation by the CODEX program [3], which is essentially the same as for the FORTRAN compiler [4]. The CODEX program, which was developed at the Mathematics Research Center, prepares the subroutines for the evaluation of the  $n$  functions  $f_i$ ,  $i = 1, 2, \dots, n$ , and the  $n^2$  derivatives  $\partial f_i / \partial x_j$ ,  $i, j = 1, 2, \dots, n$ . This relieves the user of a tedious chore, and removes a possible source of error. This takes care of (1) and (2).

During the operation of CODEX, one of the following error messages may be printed if the corresponding restriction is violated.

1. "PARENTHESIS ERROR IN DEFINITION OF name of function." Check the parentheses in the function named, correct the error and resubmit.

2. "STORAGE INSUFFICIENT FOR COMPILING." This message signifies that the system of equations is too large for the program to handle. At present, the program will handle a system of 24 equations in 24 unknowns. The equations are relatively sparse, however, and there is no guarantee that another system of that size could be handled. Unfortunately, the program occupies most of the storage available in the CDC 3600, so little can be done outside of rewriting the

entire program when this message is received.

3. "Name of storage area STORAGE INSUFFICIENT FOR DIFFERENTIATION." This message occurs when storage is exceeded during differentiation of the equations. As above, there is not much that can be done.

4. "Name NOT DIFFERENTIABLE." This is caused by attempting to differentiate an operator whose derivative has not been defined. Check the equations.

5. "ILLEGAL VARIABLE DETECTED." This occurs when the evaluating portion of the program comes across an improperly named variable. Check for a variable whose name has more than 3 characters.

The matrix operations (3) are standard, and the user may employ any matrix inversion program he chooses, provided that it gives the indication of failure

$$\text{ISING} \neq 0 \quad (3.3)$$

on return to the main program. The routine used here (INVERT) is a slow but accurate program which uses double pivoting. For large nonlinear systems, such as arise in the solution of nonlinear elliptic boundary value problems [5], an iterative subroutine may be required.

If the matrix inversion fails, the program terminates by printing the message

"DIVERGENCE INDICATED AT ITERATION NUMBER \_\_\_\_\_ DUE TO  
FAILURE OF MATRIX INVERSION."

The matrix which the inversion subroutine failed to invert will be printed if the user desires. A new value of  $x_0$  may be taken by the program at this point.

The following constants are calculated by NEWTON for comparison with tolerances provided by the user:

(1)  $\|F(x_0)\|$  is compared to the given numbers  $F$  and  $FF$ . If

$$\|F(x_0)\| \leq F, \quad (3.4)$$

then  $x^* = x_0$  (the current value of  $x$ ) is taken to be a solution of (2.1).

Following the message

"SUCCESSFUL CONVERGENCE AT ITERATION NUMBER \_\_\_\_\_ WITH

NORMF = \_\_\_\_\_ LESS THAN OR EQUAL TO \_\_\_\_\_."

the values of  $\xi_1^*, \xi_2^*, \dots, \xi_n^*$  and  $f_1(\xi_1^*, \xi_2^*, \dots, \xi_n^*)$ ,

$f_2(\xi_1^*, \xi_2^*, \dots, \xi_n^*), \dots, f_n(\xi_1^*, \xi_2^*, \dots, \xi_n^*)$  are printed.

If

$$\|F(x_0)\| > FF, \quad (3.5)$$

then it is assumed that the method is divergent, and the program is stopped (or given another value of  $x_0$ ). This feature prevents generation of a sequence of useless values. The message printed in this case is

"DIVERGENCE INDICATED AT ITERATION NUMBER \_\_\_\_\_ AS NORMF

= \_\_\_\_\_ IS GREATER THAN \_\_\_\_\_."

(2) The number

$$\|G\| = \|[F'(x_0)]^{-1}\| \quad (3.6)$$

is compared to the given number  $BB$ . If

$$\|G\| > BB, \quad (3.7)$$

then the program terminates the iteration and prints the message

"DIVERGENCE INDICATED AT ITERATION NUMBER \_\_\_\_\_ AS

BOUND  $G$  = \_\_\_\_\_ IS GREATER THAN \_\_\_\_\_."



If the user desires, the matrix  $G = [F'(x_0)]^{-1}$  will also be printed. Condition (3.7) is used for a divergence criterion for two reasons: A large value for  $\|G\|$  indicates that  $F'(x_0)$  may be singular or nearly singular, hence the components of  $G$  may be in error by large amounts; also, the value of  $x_1$  will be inaccurate even if  $F(x_0)$  is known fairly exactly.

(3) The quantity

$$\|x_1 - x_0\| = \|-[F'(x_0)]^{-1}F(x_0)\|, \quad (3.8)$$

is compared to the given numbers  $C$  and  $CC$ . If

$$\|x_1 - x_0\| \leq C, \quad (3.9)$$

then  $x^* = x_1$  is taken to be a solution of (2.1). The message,

"SUCCESSFUL CONVERGENCE AT ITERATION NUMBER \_\_\_\_ WITH  
NORMCX = \_\_\_\_ LESS THAN OR EQUAL TO \_\_\_\_,"

is printed, followed by the values of  $x^*$  and  $F(x^*)$ . If

$$\|x_1 - x_0\| > CC, \quad (3.10)$$

divergence is assumed, and the program terminates the iteration and prints the message:

"DIVERGENCE INDICATED AT ITERATION NUMBER \_\_\_\_ AS  
NORMCX = \_\_\_\_ IS GREATER THAN \_\_\_\_."

(4) The total number of iterations  $m$  is compared with the given number LIMIT. If

$$m > \text{LIMIT}, \quad (3.11)$$

divergence is assumed, the iteration terminates, and the message,

"DIVERGENCE INDICATED AT ITERATION NUMBER \_\_\_\_\_ AS THE  
NUMBER OF ITERATIONS HAS EXCEEDED \_\_\_\_\_ ,"

printed. This control prevents the computer from generating a sequence which flounders aimlessly.

(5) Finally, each iteration is timed, and the total elapsed time plus the time for the previous iteration is compared to TLIM , the number of milliseconds allowed for the total iteration by the user. If this estimate for total time at the end of the next iteration exceeds TLIM , the program prints the message:

"NOT ENOUGH TIME REMAINS FOR THE NEXT ITERATION, "

and the current values of  $x$  ,  $F(x)$  , and other parameters. This feature of the program prevents loss of information due to a time limit interrupt.

4. Error Estimation. An optional feature of NEWTON is automatic error estimation, using (2.28), which may be written

$$\|x^* - x_1\| \leq 2 B_0 \eta_0^2 K . \quad (4.1)$$

The quantities

$$B_0 = \|[F'(x_0)]^{-1}\| , \quad \eta_0 = \|x_1 - x_0\| \quad (4.2)$$

are available immediately from the computation of  $x_1$  by the process described in Section 3. The only remaining quantity is the bound

$$K \geq \|F''(x)\| \quad (4.3)$$

in a ball  $V(x_0, r)$  ,

$$V(x_0, r) = \{x : \|x - x_0\| \leq r\} \quad (4.4)$$

of sufficiently large radius  $r$  so that (2.26) will be satisfied if  $h_0 \leq \frac{1}{2}$ .

Two options are available to the user:

(1) If a value of  $K$ , or a special method for computing  $K$  is known, then this value, or a subroutine for computing  $K$ , may be inserted into the program. The value of  $K$  is called BNORM in the program.

(2) The program will form the second derivatives required for  $F''(x)$  as given by (2.6), and estimate  $K$  by the use of interval arithmetic [6,7]. This estimation makes use of the program INTERVAL [8], which was developed at the Mathematics Research Center to add interval arithmetic to the modes of computation available on the CDC 1604 and CDC 3600.

To perform this estimation, subroutines for the evaluation of the  $n(n-1)/2$  distinct second derivatives (2.6) in interval arithmetic are compiled by CODEX and INTERVAL. (Recall that

$$\frac{\partial^2 f_i}{\partial \xi_1 \partial \xi_j} = \frac{\partial^2 f_i}{\partial \xi_j \partial \xi_1}, \quad (4.5)$$

$i, j, k = 1, 2, \dots, n$ .) Each derivative

$$g_{ijk}(x) = g_{ijk}(\xi_1, \xi_2, \dots, \xi_n) = \frac{\partial^2 f_i}{\partial \xi_1 \partial \xi_j} \quad (4.6)$$

is evaluated as an interval-valued function of the interval vector

$$X^{(0)} = (\Xi_1^{(0)}, \Xi_2^{(0)}, \dots, \Xi_n^{(0)}) \quad (4.7)$$

with components which are the intervals

$$\Xi_1^{(0)} = [\xi_1^{(0)} - 2\eta_0, \xi_1^{(0)} + 2\eta_0] \quad (4.8)$$

Thus, the vectors  $x = (\xi_1, \xi_2, \dots, \xi_n)$  belonging to  $X^{(0)}$  lie in the ball  $V(x_0, 2\eta_0)$ , in fact, by (2.9) and (4.4),

$$X^{(0)} = V(x_0, 2\eta_0) . \quad (4.9)$$

As a function of  $X^{(0)}$ , the interval value of  $\frac{\partial^2 f_i}{\partial \xi_i \partial \xi_j}$  will be called

$G_{ijk}(X^{(0)})$ , and for

$$G_{ijk}(X^{(0)}) = [a, b] , \quad (4.10)$$

one has that

$$a \leq \min_{x \in X^{(0)}} g_{ijk}(x) , \quad \max_{x \in X^{(0)}} g_{ijk}(x) \leq b , \quad (4.11)$$

by the fundamental theorem of interval arithmetic [6, 7], and thus

$$|b_{ijk}| = \max_{x \in X^{(0)}} |g_{ijk}(x)| \leq \max\{|a|, |b|\} . \quad (4.12)$$

Therefore, by (2.11),

$$K \leq \max_{(i)} \sum_{j=1}^n \sum_{k=1}^n |b_{ijk}| . \quad (4.13)$$

The quantity on the right is a rigorous, but usually gross, upper bound for  $K$  in the ball  $V(x_0, 2\eta_0)$ . If the INTERVAL program is unable to compute finite real values of  $[a, b]$  for any function or derivative, then it will print the appropriate error message [8]. The NEWTON computation is aborted in this case.

Using the value for  $K$  resulting from either option described above, the number

$$h_0 = B_0 \eta_0 K \quad (4.14)$$

is calculated, and compared to  $\frac{1}{2}$  and the given number  $HH$ . If

$$h_0 > HH , \quad (4.15)$$

The iteration is assumed to be divergent, and is terminated. This situation is indicated by the message

"DIVERGENCE INDICATED AT ITERATION NUMBER \_\_\_\_\_ AS

$H_0 = \underline{\hspace{1cm}}$  IS GREATER THAN  $\underline{\hspace{1cm}}$ ."

If

$$h_0 \leq HH, \quad (4.16)$$

then the error bound (2.28),

$$\|x^* - x_1\| \leq 2h_0 \eta_0 \quad (4.17)$$

is calculated, and compared to preassigned number  $E$ . If

$$\|x^* - x_1\| \geq E, \quad (4.18)$$

then the program performs another iteration with  $x_0 := x_1$ . If

$$\|x^* - x_1\| < E, \quad (4.19)$$

then  $x_1$  is regarded as being a sufficiently accurate approximation to the solution of  $F(x) = 0$ , and the program prints the message:

"SUCCESSFUL CONVERGENCE AT ITERATION NUMBER \_\_\_\_\_ WITH

ERROR =  $\underline{\hspace{1cm}}$  LESS THAN  $\underline{\hspace{1cm}}$ ,"

followed by the values of  $x_1$  and  $F(x_1)$ .

During the operation of the program while using the error estimation option, all of the controls described in Section 3 remain in effect. The automatic error estimation feature, using INTERVAL, lengthens the computation time for each iteration considerably. In the case of a simple system of three equations in three unknowns to be presented later as an example, this amounts to a factor of ten. Consequently, unless an error estimate is of great moment, one of the other parameters could be taken as an accuracy control, perhaps after a test run

on a typical case with error estimation shows that some other criterion is reliable. Because of the rapid convergence of Newton's method, as shown by (2.27), the price of an extra iteration or two to obtain a value of  $\|x_1 - x_0\|$  or  $\|F(x_1)\|$  which is smaller than necessary for the required accuracy is probably less than that of the automatic error estimation procedure.

It is, of course, possible to become fanatical about rigorous error estimation. One may note that the computation of  $F(x_0)$ ,  $F'(x_0)$ ,  $[F'(x_0)]^{-1}$ , and thus  $x_1$ , are subject to round-off error, so that one obtains some  $\tilde{x}_1$  instead of the  $x_1$  called for by the theory in Section 2. If one can estimate  $\|x_1 - \tilde{x}_1\|$  by interval methods [6,7] or by other procedures [9,10], then

$$\|x^* - \tilde{x}_1\| \leq \|x_1 - \tilde{x}_1\| + \|x^* - x_1\| \leq 2h_0\eta_0 + \|x_1 - \tilde{x}_1\| \quad (4.20)$$

is a rigorous bound, as long as  $B_0, \eta_0, K$ , and thus  $h_0$ , are upper bounds for the corresponding exact quantities. (In using automatic error analysis, the factor of overestimation of  $K$  usually dominates the much smaller errors in the calculation of  $x_1$ , so that (4.17) gives a correct, if pessimistic, result.)

In addition, there can be errors in the coefficients of the system to be solved, or limitation on the accuracy with which they are known. This gives rise to an uncertainty error in the numerical solution. Also, if the system to be solved is a finite approximation to a differential or integral equation, there is a discretization error due to the method of approximation used. Analysis of these errors is completely outside the scope of this paper.

5. Flow chart. The structure of the program, which was described above in narrative fashion, is shown geometrically by the flow-chart in figures 2, 3, and 4.

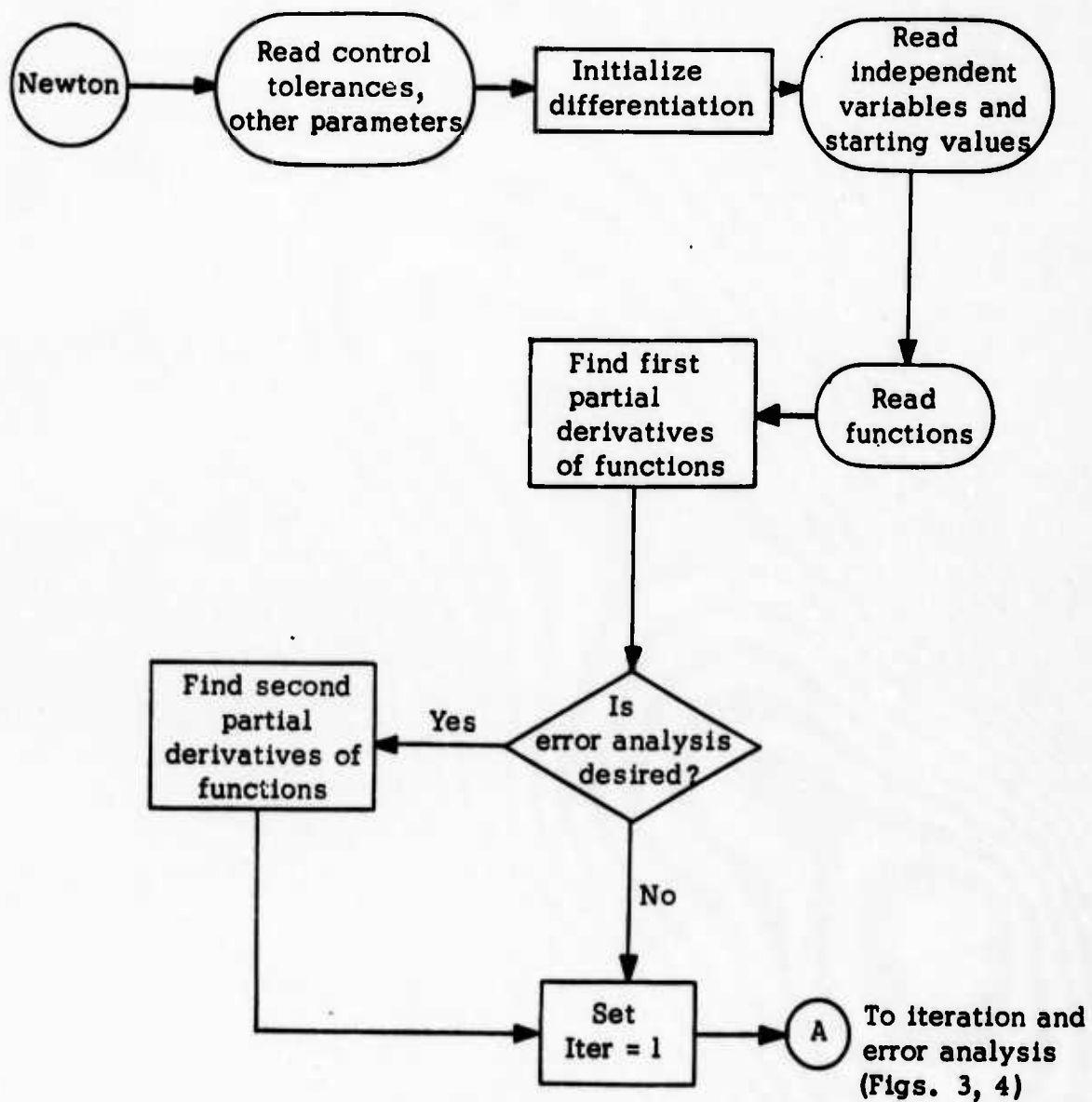


Figure 2. Initialization



From initialization  
(Fig. 2)

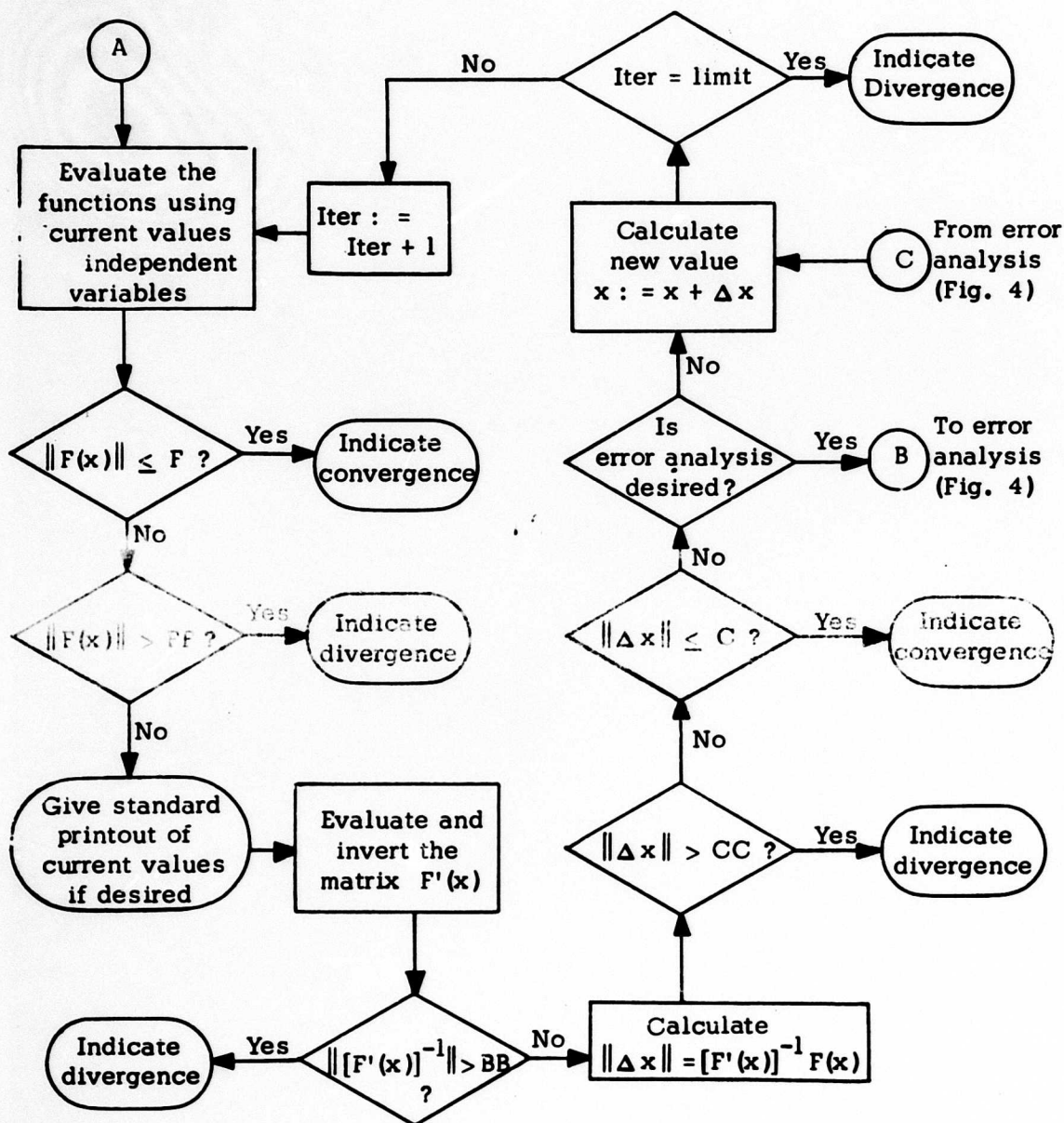


Figure 3. Iteration



From iteration  
(Fig. 3)

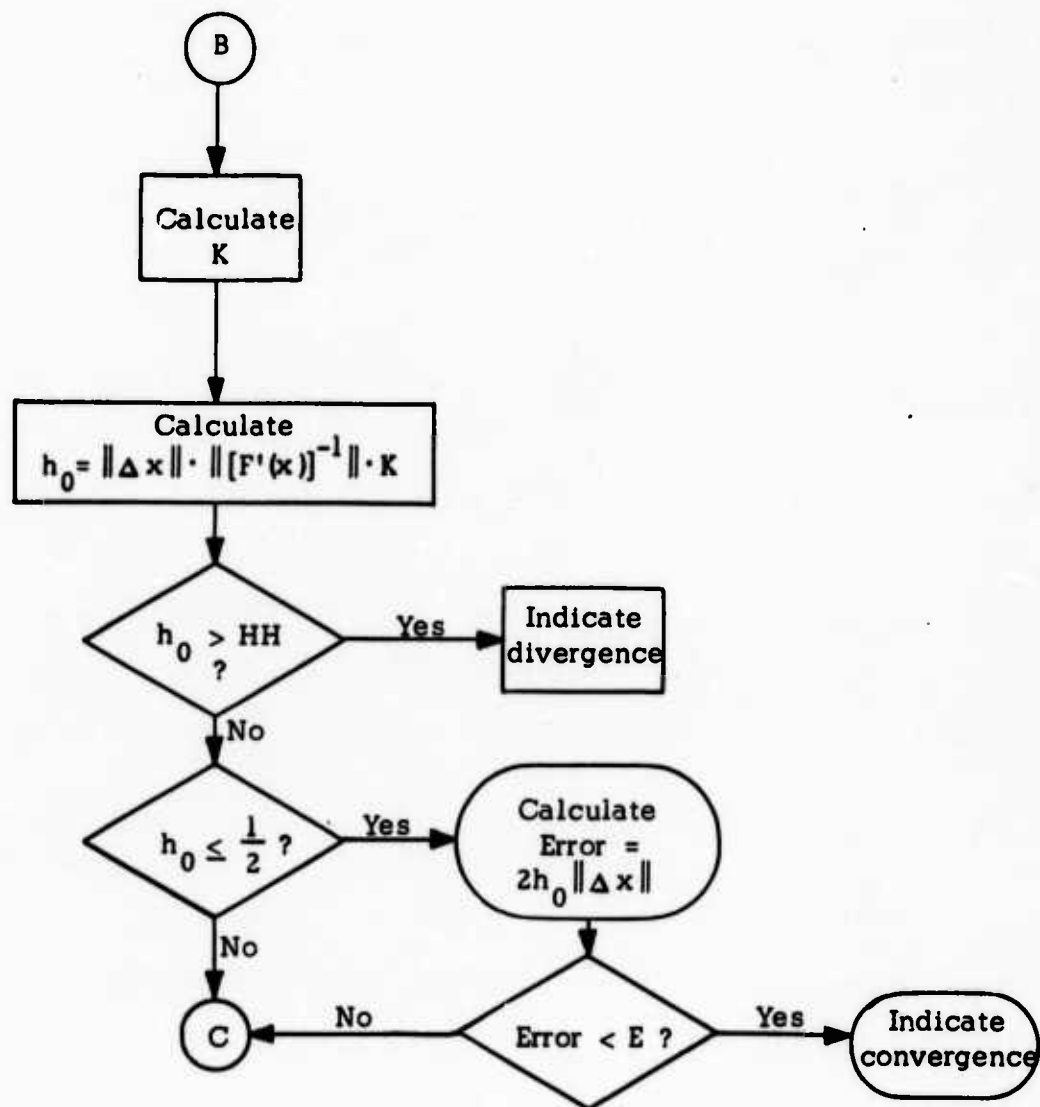


Fig. 4. Error Analysis

6. Input. Explanations of format designations may be found in [4].

The first input card contains, in I5 format, the number of systems of equations to be read in and solved during the run. The second input card, which is read in under 3E20.8 format, contains in columns 01-20,  $F$ , the value of the tolerance to be allowed on  $\|F(x_n)\|$ . Columns 21-40 of the second card contain  $FF$  the upper bound for  $\|F(x_n)\|$ . Columns 41-60 of the second input card contain  $BB$ , the upper bound on the norm of the inverse of the partial equations matrix.

The third input card, also read in under 3E20.8 format, contains in the first 20 columns  $C$ , the tolerance on the norm of the increment vector. Columns 21-40 of the third input card contain  $CC$ , the upper bound on the norm of the increment vector. Columns 41-60 of the third input card contain the time in milliseconds allowed for the iteration section of the program.

The next input card supplies the program with parameters which will determine what options are to be used as well as several iteration limits. The card is read in under a I5I5 format. If columns 1-5 are zero, the error analysis subroutine will not be used, otherwise, the subroutine will be called. Columns 6-10 indicate whether or not the matrix of the partial derivatives is to be printed out in case the inversion of this matrix fails. If columns 6-10 are zero, the matrix will not be printed out, otherwise, a printout will be given. If columns 11-15 are not zero, a printout of the inverse of the Jacobian matrix will be given when the norm of this matrix exceeds the given upper bound and divergence of the system is thus indicated. If columns 11-15 are zero, no printout will be given. Columns 16-20 indicate how often a printout of the intermediate

values of the Newton sequence is desired. If this printout is desired every time, then there should be a 1 in column 20, if every other time, column 20 should be 2, etc. Columns 21-25 give the number of iterations to be allowed in searching for a solution. This number must be right adjusted in the field. Columns 26-30 give the number of sets of starting values which are to be used with the system of equations. If columns 36-40 are zero, there will be a printout of the formulas which CODEX makes up for the given equations, the partial derivatives, and the second partial derivatives. If columns 36-40 are not zero, no printout will be given. Columns 41-45 need be used only if the error routine is being used. If so, then column 45 is 1 if the norm of the matrix of the second partial derivatives is not known and must be computed. If the norm is known, then column 45 is 2.

The next input card is supplied only if the error analysis routine is to be used. Otherwise, it should not be present. This card is read in under 3E20.8 format. Columns 1-20 of this card contain the allowed tolerance E on the error bound. Columns 21-40 contain the upper bound HH on the convergence constant. Columns 41-60 need be supplied only when the norm K of the second derivative is known. These columns then contain this norm.

The next data cards contain the names of the independent variables and their starting values. The names of the variables are limited to three non-blank alphanumeric characters, the first of which must be an alphabetic character. The first name may be punched in any of the first 72 columns of the card. It is followed by at least one blank and the starting value corresponding to the variable. The starting value must be followed by at least one blank, and then the name of

the next independent variable and its starting value are given. When all of the independent variables and their starting values have been given, the last entry is followed by at least one blank and a \$. The starting values may be given as a fixed point integer, a floating-point number with a decimal point, or a FORTRAN E-format number. The numbers may be signed or unsigned.

The last group of data cards contain the equations for which a solution is to be found. These must be in the form  $F(x) = 0$  where  $F(x)$  is an arithmetic expression using any of the operations  $+$ ,  $-$ ,  $*$ ,  $/$ ,  $**$  and/or any of the transcendental functions  $\text{sine}(x)$ ,  $\text{SINF}(x)$ ,  $\text{cosine}(x)$ ,  $\text{COSF}(x)$ , (natural)  $\log(x)$ ,  $\text{LOGF}(x)$ ,  $\exp(x)$ ,  $\text{EXPF}(x)$ , and  $\text{arctangent}(x)$ ,  $\text{ATAN}(x)$ .  $F(x)$  is then given to the program in the form

$$\text{variable name} = F(x) \quad .$$

The above formula must be punched with at least one blank between consecutive symbols. As with the independent variables, only the first 72 columns of a card are significant. The formula may be continued on any number of consecutive cards and is terminated by a blank and a \$ following the last symbol of  $F(x)$ .

The program reads in the independent variables until it encounters a \$. Then it expects to read in as many equations as independent variables, and it separates and counts these by the \$ at the end. If a \$ is misplaced, any of error returns 1, 4, and 5 from CODEX, as well as an unchecked EOF are equally likely to occur.

7. An example. The results of computation of the solution of the system

$$\begin{aligned}
 16x^4 + 16y^4 + z^4 - 16 &= 0 \\
 x^2 + y^2 + z^2 - 3 &= 0 \\
 x^3 - y &= 0
 \end{aligned}
 \tag{6.1}$$

in the first octant are shown in Appendix I with and without automatic error estimation. The initial approximation was taken to be

$$x_0 = (1, 1, 1) \quad . \tag{6.2}$$

Other applications of this program have been made to finding characteristic values and vectors of matrices [11,12], and solutions of systems arising in magnetohydrodynamic problems [13]. Its performance in every case has been satisfactory.

8. Warning. The complete program, except for unmodified subroutines of CODEX [3] and INTERVAL [8] is listed in Appendix II. Many subroutines are in CDC 3600 machine language [14], or use constants peculiar to the CDC 3600. Consequently, it is doubtful that the program as listed will work at any other installation, or survive future changes in the operating systems program at the University of Wisconsin Computing Center. However, the listing given, together with the description given above, should be a reliable guide for the adaptation of this program for use elsewhere.

## REFERENCES

1. Kantorovič, L. V. [Kantorovich, L. V.]. Functional analysis and applied mathematics. Tr. by C. D. Benster. National Bur. Standards Report No. 1509, Los Angeles, 1952.
2. Rall, L. B. Computational solution of nonlinear operator equations. To be published by John Wiley & Sons, New York, 1968.
3. Gray, Julia H., and Reiter, Allen. CODEX: Compiler of differentiable functions. Mathematics Research Center, U. S. Army, Technical Summary Report No. 791. Madison, 1967.
4. CDC 3600 FORTRAN/Reference Manual Preliminary. Control Data Corp., Minneapolis, 1964.
5. Greenspan, D. Introductory numerical analysis of elliptic boundary value problems. Harper and Row, New York, 1965.
6. Moore, R. E. The automatic analysis and control of error in digital computing based on the use of interval numbers. Error in Digital Computation, Vol. 1, pp. 61-130. L. B. Rall, editor. John Wiley & Sons, New York, 1965.
7. Moore, R. E. Interval analysis. Prentice-Hall, Englewood Cliffs, N.J., 1966.
8. Reiter, Allen. Interval arithmetic package (INTERVAL) for the CDC 1604 and the CDC 3600. Mathematics Research Center, U. S. Army, Technical Summary Report No. 794. Madison, 1967.
9. Albasiny, E. L. Error in digital solution of linear problems. Error in Digital Computation, Vol. 1, pp. 131-184. L. B. Rall, editor, John Wiley & Sons, New York, 1965.
10. Wilkinson, J. H. Rounding errors in algebraic processes, Prentice-Hall, New York, 1963.
11. Rall, L. B. Newton's method for the characteristic value problem  $Ax = \lambda x$ . J. Soc. Indust. Appl. Math. 9 (1961), 288-293. Errata 10 (1962), 228.
12. Anselone, P. M., and Rall, L. B. Newton's method for characteristic value-vector problems. Mathematics Research Center, U. S. Army, Technical Summary Report No. 492, Madison, 1964.
13. Pack, D. C., and Swan, G. W. Magneto-gasdynamics flow over a wedge. J. Fluid Mech. 25 (1966), 165-178.
14. CDC 3600 Reference Manual. Control Data Corporation, Minneapolis, 1964.

# APPENDIX I

(1) Example without automatic error estimation.

F =	1.00000000-009
FF =	1.00000000+006
BB =	1.00000000+006
C =	1.00000000-009
CC =	1.00000000+006
TLIM =	1.20000000+005
IERR =	0 IMAT = 1 IDMP = 1
NITDIS =	1 LIMIT = 25 IAGAIN = 1
NOF =	1 NPRT = 0 IAVL = 1

## NEWTONS METHOD

X 1. Y 1. Z 1. S

F1 = 16. \* X \*\* 4 + 16. \* Y \*\* 4 + Z \*\* 4 - 16. S

### CODE LIST FOR F1

0001T = X \*\* 0004C

0002T = 0012C \* 0001T

0003T = Y \*\* 0004C

0004T = 0012C \* 0003T

0005T = 0002T + 0004T

0006T = Z \*\* 0004C

0007T = 0005T + 0006T

F1 = 0007T - 0012C

F2 = X \*\* 2 + Y \*\* 2 + Z \*\* 2 - 3. S

### CODE LIST FOR F2

0010T = X \*\* 0002C

0011T = Y \*\* 0002C

0012T = 0010T + 0011T

0013T = Z \*\* 0002C

0014T = 0012T + 0013T

F2 = 0014T - 0003C

F3 = X \*\* 3 - Y S

### CODE LIST FOR F3

0015T = X \*\* 0003C

F3 = 0015T - Y



CODE LIST FOR 1, 1, 0			
0016T	=	X	** 0003C
0017T	=	0004C	* 0010T
0020T	=	0012C	* 0017T
011	=		+ 0020T

CODE LIST FOR 1, 2, 0			
0021T	=	Y	** 0003C
0022T	=	0004C	* 0021T
0023T	=	0012C	* 0022T
012	=		+ 0023T

CODE LIST FOR 1, 3, 0			
0024T	=	Z	** 0003C
0025T	=	0004C	* 0024T
013	=		+ 0025T

CODE LIST FOR 2, 1, 0			
0026T	=	0002C	* X
021	=		+ 0026T

CODE LIST FOR 2, 2, 0			
0027T	=	0002C	* Y
022	=		+ 0027T

CODE LIST FOR 2, 3, 0			
0030T	=	0002C	* Z
023	=		+ 0030T

CODE LIST FOR 3, 1, 0			
0031T	=	X	** 0002C
0032T	=	0003C	* 0031T
031	=		+ 0032T

CODE LIST FOR 3, 2, 0			

032	=		- 0001C
-----	---	--	---------

CODE LIST FOR 3, 3, 0			
033	NOT ON F-LIST.		

ITERATION NUMBER 1

NORM F = 1.70000000+001

X = 1.00000000+000

Y = 1.00000000+000

Z = 1.00000000+000

F1 = 1.70000000+001

F2 = 0.00000000+000

F3 = 0.00000000+000

TIME PER ITERATION = 70.00 MILLISECONDS

ITERATION NUMBER 2

NORM F = 4.79191714+000

NORM CX = 2.83333333-001

BOUND FPRIME INVERSE = 5.50000000-001

X = 9.29166667-001

Y = 7.87500000-001

Z = 1.28333333+000

F1 = 4.79191714+000

F2 = 1.30451389-001

F3 = 1.46966869-002

TIME PER ITERATION = 125.00 MILLISECONDS

ITERATION NUMBER 3

NORM F = 6.45309522-001

NORM CX = 9.43241407-002

BOUND FPRIME INVERSE = 5.56439198-001

X = 8.87074529-001

Y = 6.93175859-001

Z = 1.32086464+000

F1 = 6.45309522-001

F2 = 1.20773905-002

F3 = 4.86417094-003

TIME PER ITERATION = 480.00 MILLISECONDS

ITERATION NUMBER 4

NORM F = 1.84509452-002

NORM CX = 1.59811520-002

BOUND FPRIME INVERSE = 6.34254336-001

X = 8.78244398-001

Y = 6.77194707-001

Z = 1.33060980+000

F1 = 1.84509452-002

F2 = 4.28336556-004

F3 = 2.06810364-004

TIME PER ITERATION = 132.00 MILLISECONDS

ITERATION NUMBER 5

NORM F = 1.47977844-005

NORM CX = 4.37403606-004

BOUND FPRIME INVERSE = 6.61437802-001

X = 8.77965993-001

Y = 6.76757304-001

Z = 1.33085521+000

F1 = 1.47977844-005

F2 = 3.29047907-007

F3 = 2.04207026-007

TIME PER ITERATION = 127.00 MILLISECONDS

SUCCESSFUL CONVERGENCE AT ITERATION NUMBER 6

WITH NORMF = 4.65661287-010 LESS THAN OR EQUAL TO 1.00000000-009

NORM CX	=	3.33198155-007	BOUND FPRIME INVERSE	=	6.62185787-001
X	=	8.77965760-001	F1	=	4.65661287-010
Y	=	6.76756971-001	F2	=	5.82076609-011
Z	=	1.33085541+000	F3	=	0.00000000+000

(2) Example with automatic error estimation.

```

F = 1.00000000-009
FF = 1.00000000+006
BB = 1.00000000+006
C = 1.00000000-009
CC = 1.00000000+006
TLIM = 1.20000000+005
IERR = 1 IMAT = 1 IDMP = 1
NITCIS = 1 LIMIT = 25 IAGAIN = 1
NCF = 1 NPRT = 0 IAVL = 1
E = 1.00000000-008
HH = 1.00000000+006
BNCRM = -0.00000000+000

```

NEWTONS METHOD

X 1. Y 1. Z 1. S

F1 = 16. \* X \*\* 4 + 16. \* Y \*\* 4 + Z \*\* 4 - 16. S

CODE LIST FOR F1

```

0001T = X ** 0004C
0002T = 0012C * 0001T
0003T = Y ** 0004C
0004T = 0012C * 0003T
0005T = 0002T + 0004T
0006T = Z ** 0004C
0007T = 0005T + 0006T
F1 = 0007T - 0012C

```

F2 = X \*\* 2 + Y \*\* 2 + Z \*\* 2 - 3. S

CODE LIST FOR F2

```

0010T = X ** 0002C
0011T = Y ** 0002C
0012T = 0010T + 0011T
0013T = Z ** 0002C
0014T = 0012T + 0013T
F2 = 0014T - 0003C

```

F3 = X \*\* 3 - Y S

CODE LIST FOR F3

```

0015T = X ** 0003C
F3 = 0015T - Y

```

CODE LIST FOR 1, 1, 0			
0016T	=	X	** 0003C
0017T	=	0004C	* 0016T
0020T	=	0012C	* 0017T
011	=		+ 0020T

CODE LIST FOR 1, 2, 0			
0021T	=	Y	** 0003C
0022T	=	0004C	* 0021T
0023T	=	0012C	* 0022T
012	=		+ 0023T

CODE LIST FOR 1, 3, 0			
0024T	=	Z	** 0003C
0025T	=	0004C	* 0024T
013	=		+ 0025T

CODE LIST FOR 1, 1, 1			
5671T	=	X	** 0002C
5672T	=	0003C	* 5671T
5673T	=	0004C	* 5672T
5674T	=	0012C	* 5673T
111	=		+ 5674T

CODE LIST FOR 1, 1, 2			
112 NOT ON F-LIST.			

CODE LIST FOR 1, 1, 3			
113 NOT ON F-LIST.			

CODE LIST FOR 1, 2, 2			
5671T	=	Y	** 0002C
5672T	=	0003C	* 5671T
5673T	=	0004C	* 5672T
5674T	=	0012C	* 5673T
122	=		+ 5674T

CODE LIST FOR 1, 2, 3			
123 NOT ON F-LIST.			

CODE LIST FOR 1, 3, 3			
5671T	=	Z	** 0002C
5672T	=	0003C	* 5671T
5673T	=	0004C	* 5672T
133	=		+ 5673T

CODE LIST FOR 2, 1, 0

0026T = 0002C \* X  
021 = + 0026T

CODE LIST FOR 2, 2, 0

0027T = 0002C \* Y  
022 = + 0027T

CODE LIST FOR 2, 3, 0

0030T = 0002C \* Z  
023 = + 0030T

CODE LIST FOR 2, 1, 1

211 = + 0002C

CODE LIST FOR 2, 1, 2

212 NOT ON F-LIST.

CODE LIST FOR 2, 1, 3

213 NOT ON F-LIST.

CODE LIST FOR 2, 2, 2

222 = + 0002C

CODE LIST FOR 2, 2, 3

223 NOT ON F-LIST.

CODE LIST FOR 2, 3, 3

233 = + 0002C

CODE LIST FOR 3, 1, 0

0031T = X \*\* 0002C  
0032T = 0003C \* 0031T  
031 = + 0032T

CODE LIST FOR 3, 2, 0

032 = - 0001C

CODE LIST FOR 3, 3, 0

033 NOT ON F-LIST.

CODE LIST FOR 3, 1, 1

5671T = 0002C \* X  
5672T = 0003C \* 5671T  
311 = + 5672T

CODE LIST FOR 3, 1, 2

312 NOT ON F-LIST.

CODE LIST FOR 3, 1, 3

313 NOT ON F-LIST.

CODE LIST FOR 3, 2, 2

322 NOT ON F-LIST.

CODE LIST FOR 3, 2, 3

323 NOT ON F-LIST.

CODE LIST FOR 3, 3, 3

333 NOT ON F-LIST.

ITERATION NUMBER 1			
NORM F = 1.70000000+001			
X	=	1.00000000+000	F1 = 1.70000000+001
Y	=	1.00000000+000	F2 = 0.00000000+000
Z	=	1.00000000+000	F3 = 0.00000000+000

TIME PER ITERATION = 64.00 MILLISECONDS

ITERATION NUMBER 2			
NORM F = 4.79191714+000			
NORM CX = 2.83333333-001			
BOUND FPRIME INVERSE = 5.50000000-001			
BOUND F DRL PRIME = 9.71960000+002			
HC = 1.51463767+002			
X	=	9.29166667-001	F1 = 4.79191714+000
Y	=	7.87500000-001	F2 = 1.30451389-001
Z	=	1.28333333+000	F3 = 1.46966869-002

TIME PER ITERATION = 1928.00 MILLISECONDS

ITERATION NUMBER 3			
NORM F = 6.45309522-001			
NORM CX = 9.43241407-002			
BOUND FPRIME INVERSE = 5.56439198-001			
BOUND F DRL PRIME = 4.43856898+002			
HC = 2.35585457+001			
X	=	8.87074529-001	F1 = 6.45309522-001
Y	=	6.93175859-001	F2 = 1.20773905-002
Z	=	1.32086464+000	F3 = 4.86417094-003

TIME PER ITERATION = 1843.00 MILLISECONDS

ITERATION NUMBER 4			
NORM F = 1.84509452-002			
NORM CX = 1.59811520-002			
BOUND FPRIME INVERSE = 6.34254336-001			
BOUND F DRL PRIME = 2.85088868+002			
HC = 2.88969353+000			
X	=	8.78244398-001	F1 = 1.84509452-002
Y	=	6.77194707-001	F2 = 4.28336556-004
Z	=	1.33060980+000	F3 = 2.06810364-004

TIME PER ITERATION = 1767.00 MILLISECONDS



ITERATION NUMBER		5
<hr/>		
NORM F =	1.47977844-005	
NORM CA =	4.37403606-004	
<hr/>		
BOUND FPRIME INVERSE =	6.61437802-001	
BOUND F DBL PRIME =	2.57938953+002	
HO =	7.46256805-002	
ERRCR =	6.52830835-005	
X =	8.77965993-001	F1 = 1.47977844-005
Y =	6.76757304-001	F2 = 3.29047907-007
Z =	1.33085521+000	F3 = 2.04207026-007
<hr/>		
TIME PER ITERATION =	1853.00 MILLISECONDS	

SUCCESSFUL CONVERGENCE AT ITERATION NUMBER 5

WITH ERROR = 3.79255563-011 LESS THAN		1.00000000-008
X =	8.77965760-001	F1 = 4.65661287-010
Y =	6.76756971-001	F2 = 5.82076609-011
Z =	1.33085541+000	F3 = 0.00000000+000

## APPENDIX II

Listing of NEWTON and Relevant Subroutines (July, 1967), not including  
CODEX and INTERVAL.

```

PROGRAM N
  DIMENSION P(24,24),B(24),CX(24)
  DIMENSION TSTORE(8190)
  COMMON/ENAMES/KE,LE(700),LS(700)
  COMMON/CONST/CC,LCOM(4005),LCVM(4005)/LI/LIRFJN(10)
  COMMON/VALUES/M(700)
  COMMON/KTST/KTST
  COMMON LXY(5000),KXY
  COMMON/ERROR/PPINVN,CX1,E,HH,BNORM,H0,ERR,NSYS
  EQUIVALENCE (LXY,TSTORE)
  REWIND 47
  READ 1,NCASES
  FORMAT (15I5)
  DO 500 NC=1,NCASES
    STIME=TIMEF(DUMMY)
    READ 2,F,FF,RR,C,CC,TLIM
    FORMAT (3E20.8)
    READ 1,IFRR,IMAT,IDMP,NITDIS,LIMIT,IAGAIN,NOF,NPRT,IAVL1
    PRINT 6,F,FF,RR,C,CC,TLIM
    FORMAT(5H F = F16.8, /6H FF = F16.8, /6H RR = F16.8, /5H C = F16.8
    , /6H CC = F16.8, /8H TLIM = F16.8)
    PRINT 7,IFRR,IMAT,IDMP,NITDIS,LIMIT,IAGAIN,NOF,NPRT,IAVL1
    FORMAT(8H IFRR = I5, 8H IMAT = I5, 8H IDMP = I5, /10H NITDIS = I5,
    9H LIMIT = I5, 10H IAGAIN = I5, / 8H NOF = I5, 8H NPRT = I5,
    8H IAVL = I5 )
    IF(IERR)5,10
    READ 2,F,HH,BNORM1
    PRINT 8, F,HH,BNORM1
    FORMAT(5H F = F16.8, /6H HH = F16.8, / 9H BNORM = F16.8)
    DO 500 NE=1,NOF
      IDIF=1
    CALL INIT (DUMMY)
    DO 500 NX=1,IAGAIN
      READ IN INDEPENDENT VARIABLES AND STARTING VALUES
      KF=0
      PRINT 3
    
```

C C



```

KC=3000
NAME2P=(1000000B*11+100008*1J-1006*1F*60B:1C000000000b
CALL DIF(NAME2P,NAMEP,LF(IN))
IF(NPRT)560,56
CALL PRINT1(NAME2P)
IF(NAME2P)57,58
NKC=KC-3000
WRITE (47) LF(KF),NKC
WRITE (47) (LCOM(K),LCV4(K),K=3001,KC)
CONTINUE
KTST=KTS
KC=KCS
KXY=0
REWIND 47
KFS=KF
CTIME=TIMEF(DUMMY)
ITSTART=1
ITLIMIT=LIMIT
IDIV=0

MAIN BODY OF NEWTON
DO 250 ITER=ITSTART,ITLIMIT

OBTAIN P-VECTOR AND NORM
PNORM=0.
DO 101 IP=1,NSYS
CALL EVAL (LF(NSYS+IP),P(IP))
PNORM=MAX1F(PNORM,ARCF(P(IP)))
IF(PNORM.GT.F)102,300
IF(PNORM.GT.FF)400,110

STANDARD PRINTOUT
IF(XMODE(ITER,NITDTS))118,111
PRINT 1000,ITER
FORMAT(18H2 ITERATION NUMBER IS )
PRINT 1001,PNORM
FORMAT (10H0NORM F = E16.8)
IMAY=2
IF(ITER.GT.1)113,115

```

```

56
560
57
58
59
60
101
102
110
111
1000
1001

```

```

C
C
C
C
C
C
C
C
C
C

```

```

113 PPINVN=0,NSYS,PP,TSING)
1105 FORMAT (11) NORM CX E16.0, / 244 BOUND FPRIME INVERSES = 16.0 )
11 IF (ERR) 114, 115
114 PRINT 1002, P, NORM
1002 FORMAT (21H BOUND F OPL PRIME = E16.0)
1004 PRINT 1006, HC
1006 FORMAT (7H HC = F16.0)
1140 IF (IAVL.EQ.1) 115, 1140
1140 PRINT 1002, ERR
1003 FORMAT (04 ERRP = F16.0)
115 DO 116 I=1, NSYS
11 IT=NSYS+1
116 PRINT 1160, LF(I), V(I), LF(11), V(11)
1160 FORMAT (1X, A4, 3H = E16.0, 11X, A4, 3H = E16.0)
CHECK TIME
OLDTIME=CTIME
CTIME=TIMEF(SUMMY)
TIMEK=CTIME-OLDTIME
1004 PRINT 1004, TIMEK
1004 FORMAT (22H TIME PER ITERATION = F12.2, 13H MILLISECONDS )
11 IF (TLIM.GT.CTIME-SIME+TIMEK) 120, 410
120 IWAY=1
EVALUATE THE JACOBIAN MATRIX P-PRIME
DO 121 JI=1, NSYS
DO 121 JJ=1, NSYS
NAMEP=(1000000*JI+1000000*JJ+600)*10000000000
CALL EVAL (NAMEP, PP(I, JJ))
CONTINUE
121
OBTAIN P-PRIME INVERSE AND NORM
PPINVN=0.
CALL INVERT (NSYS, PP, TSING)
11 IF (TSING) 420, 122
122 DO 124 KT=1, NSYS
SUM=0.

```

```

123 DO 123 KJ=1,NSYS
124 SUM=SUM+ARCF (PP(KI,KJ))
PPINVN=MAX1F (PPINVN,SUM)
IF (PPINVN.GT.BB) 430,130
C
C
130 OBTAIN INCREMENT VECTOR AND NORM
CXN=0.
DO 132 LI=1,NSYS
CX(LI)=0.
DO 131 LJ=1,NSYS
CX(LI)=CX(LI)+P(LJ)*C(LI,LJ)
CXN=MAX1F (CXN,ABSF(CX(LI)))
IF (CXN.GT.C) 133,310
133 IF (CXN.GT.CC) 440,140
C
C
140 DO ERROR ANALYSIS IF DESIRED
IF (IERR) 141,150
141 CALL B2D (IAVL)
GO TO (150,150,320,450),IAVL
C
C
150 OBTAIN NEW SOLUTION
151 DO 151 MI=1,NSYS
151 V(MI)=V(MI)-CX(MI)
250 CONTINUE
IF (IAVL.FO.2) 270,260
C
C
260 ITERATION TIME EXCEEDED - PRINT OUT AND GO TO NEXT SET.
ITER=LIMIT+1
PRINT 2000,ITER,LIMIT
2000 FORMAT ( 42H2DIVERGENCE INDICATED AT ITERATION NUMBER IS,
1 42H AS THE NUMBER OF ITERATIONS HAS EXCEEDED 15 )
C
C
270 CALL STPRINT (NSYS)
IDIV=6
GO TO 500
ITSTART=ITER
ITLIMIT=140737488355327
271 DO 271 MI=1,NSYS
V(MI)=V(MI)-CX(MI)
GO TO 100

```





```

400  GO TO (412,500) IWAY
      CALL STPRINT(NSYS)
      GO TO 500
420  PRINT 4002,ITER
4002  FORMAT(42H2DIVERGENCE INDICATED AT ITERATION NUMBER 15,36H DUE TO
      FAILURE OF MATRIX INVERSION )
      CALL STPRINT (NSYS)
      IF(I MAT)421,500
421  PRINT 4003
4003  FORMAT(15HOF PRIME MATRIX )
      DO 423 JI=1,NSYS
      DO 422 JJ=1,NSYS
422  NAMEP=(10000R*JI+100R*JJ+60R)*1000000000B
      CALL EVAL (NAMEP,DD(JI,JJ))
      PRINT 4004,JI,(DD(JI,JJ),JJ=1,NSYS)
423  FORMAT(6H ROW 15 ( /2X,E16.8) )
4004  GO TO 500
430  PRINT 4005,ITER,PPINVN,RR
4005  FORMAT (42H2DIVERGENCE INDICATED AT ITERATION NUMBER 15,
      14H AS ROUND G = E16.8,17H IS GREATER THAN E16.8 )
      CALL STPRINT (NSYS)
      IF(I DMP)431,500
431  PRINT 4006
4006  FORMAT(30HAG = INVERSE OF F PRIME MATRIX )
      DO 432 II=1,NSYS
432  PRINT 4004,II,(DD(II,JJ),JJ=1,NSYS)
      GO TO 500
440  PRINT 4007,ITER,CXN,CC
4007  FORMAT(42H2DIVERGENCE INDICATED AT ITERATION NUMBER 15,
      13H AS NORMCX = E16.8,17H IS GREATER THAN E16.8 )
      CALL STPRINT (NSYS)
      GO TO 500
450  PRINT 4008,ITER,HO,HH
4008  FORMAT (42H2DIVERGENCE INDICATED AT ITERATION NUMBER 15,
      9H AS HO = E16.8,17H IS GREATER THAN E16.8 )
      CALL STPRINT (NSYS)
      GO TO 500
500  CONTINUE
      STOP
      END

```

```

SUBROUTINE ROUND (IAVL)
COMMON/FNAME/KF,LF(700),LS(700)/VALUES/V(700)/IVALUE/VI(1400)
COMMON/COMP/KC,LCOM(4095),LCVM(4095)
COMMON/CONST/KCON,CTAB(100)/ICONST/KCN,CTABI(200)
COMMON/ERROR/PPINVA,DXN,E,HH,BNORM,H0,ERR,NSYS
TYPE INT5 (2) DUMMY,V2P
DIMENSION V2(2),DUM(2)
EQUIVALENCE (V2,V2P),(DUM,DUMMY)
ENTRY B2D
IF(IAVL.EQ.1)100,150
100 BNORM=0.
KCN=KCON
DO 110 K=1,KCN
CTABI(2*K)=CTAB(K)
110 CTABI(2*K-1)=CTAB(K)
EPV=2.*QXN
DO 115 I=1,NSYS
VI(2*I-1)=V(I)-EPV
115 VI(2*I)=V(I)+EPV
N2=2*NSYS
DO 120 I=1,NSYS
CALL EVALINT (LF(NSYS+I),DUMMY)
120 CONTINUE
DO 125 I=1,NSYS
DO 125 J=1,NSYS
NAMEP=(I+100000B+J+100B+60B)*100000000B
CALL EVALINT (NAMEP,DUMMY)
125 CONTINUE
DO 145 I=1,NSYS
BI=0.
DO 135 J=1,NSYS
DO 135 K=J,NSYS
NAME2P=(I+1000000B+J+10000B+K+100B+60B)*100000000B
KN=NSCAN(KF,LF,NAME2P)
IF(KN.EQ.0)126,127
126 V2(1)=0.
V2(2)=0.
GO TO 128

```

```

127 READ (47),LFC,NKC
   KMX=3000+NKC
   READ (47) (LCOM(KK),LCVM(KK),KK=3001,KMX)
   IF(NAME2P.EQ.LFC)1270,200
1270 CALL EVALINT2(KN,V2P)
128 IF(K.EQ.J)129,130
129 BI=BI+FNORM(V2P)
   GO TO 135
130 BI=BI+2.*FNORM(V2P)
135 CONTINUE
   IF(BI-BNORM)145,145,140
140 BNORM=BI
145 CONTINUE
150 H0=PPINVN*DXN*BNORM
   REWIND 47
   IF(H0.GT..5)151,160
151 IF(H0.GT.HH)152,153
152 IAVL=4

153 RETURN
160 ERR=2.*DXN*H0
   IF(ERR.LT.E)165,170
165 IAVL=3
   RETURN
170 IAVL=2
   RETURN
200 PRINT 201,LFC,NAME2P
201 FORMAT (*0ERPOR ON TAPE. *016. * WAS FOUND INSTEAD OF *016 )
   END

FUNCTION FNORM (X)
  DIMENSION X(2)
  X1=ABSF(X(1))
  X2=ABSF(X(2))
  IF(X1-X2)1001,1002,1002
1001 FNORM=X2
   RETURN
1002 FNORM=X1
   END

```

C	SUBROUTINE INVERT (N,A,ISING)	CCP 4260
	DOUBLE PIVOT PROGRAM FOR MATRIX INVERSION	
	DIMENSION A(24,24),P(24,24),Q(24,24)	
	ISING=0	
	THRES =1.0E-20	CCP 4280
	NLESS1=N-1	CCP 4300
	DO 3 I=1,N	CCP 4310
	DO 3 J=1,N	CCP 4320
	IF (I-J) 1,2,1	CCP 4330
	1 P(I,J)=0.0	CCP 4340
	Q(I,J)=0.0	CCP 4350
	GO TO 3	CCP 4360
	2 P(I,J)=1.0	CCP 4370
	Q(I,J)=1.0	CCP 4380
	3 CONTINUE	CCP 4390
	DO 20 K=1,NLESS1	CCP 4400
	BIGA=0.0	CCP 4410
	KPLUS1=K+1	CCP 4420
	DO 8 I=K,N	CCP 4430
	DO 8 J=K,N	CCP 4440
	IF (A(I,J)) 4,5,5	CCP 4450
	4 ARSA=-A(I,J)	CCP 4460
	GO TO 6	CCP 4470
	5 ARSA=A(I,J)	CCP 4480
	6 IF (BIGA-ARSA) 7,8,8	CCP 4490
	7 BIGA=ARSA	CCP 4500
	LARGJ=J	CCP 4510
	LARGI=I	CCP 4520
	8 CONTINUE	CCP 4530
	IF (LARGJ-K) 25,12,9	CCP 4540
	9 DO 10 I=K,N	CCP 4550
	ASTORE=A(I,K)	CCP 4560
	A(I,K)=A(I,LARGJ)	CCP 4570
	10 A(I,LARGJ)=ASTORE	CCP 4580
	DO 11 I=1,N	CCP 4590
	QSTORE=Q(I,K)	CCP 4600
	Q(I,K)=Q(I,LARGJ)	CCP 4610
	11 Q(I,LARGJ)=QSTORE	CCP 4620
	12 IF (LARGI-K) 25,16,13	CCP 4630
	13 DO 14 J=K,N	CCP 4640

ASTORE=A(K,J)	CCP 4650
A(K,J)=A(LARGI,J)	CCP 4650
14 A(LARGI,J)=ASTORE	CCP 4670
DO 15 J=1,N	CCP 4680
PSTORE=P(K,J)	CCP 4690
P(K,J)=P(LARGI,J)	CCP 4700
15 P(LARGI,J)=PSTORE	CCP 4710
16 AMAG=ABSF(A(K,K))	CCP 4720
IF (AMAG-THRES) 24,24,17	CCP 4730
17 DO 19 I=K,NLESS1	CCP 4740
RMPY=A(I+1,K)/A(K,K)	CCP 4750
DO 18 L=KPLUS1,N	CCP 4760
18 A(I+1,L)=A(I+1,L)-RMPY*A(K,L)	CCP 4770
A(I+1,K)=0.0	CCP 4780
DO 19 LL=1,N	CCP 4790
19 P(I+1,LL)=P(I+1,LL)-RMPY*P(K,LL)	CCP 4800
DO 20 J=K,NLESS1	CCP 4810
CMPY=A(K,J+1)/A(K,K)	CCP 4820
A(K,J+1)=0.0	CCP 4830
DO 20 L=1,N	CCP 4840
20 Q(L,J+1)=Q(L,J+1)-CMPY*G(L,K)	CCP 4850
AMAG=ABSF(A(N,N))	CCP 4860
IF (AMAG-THRES) 24,24,21	CCP 4870
21 DO 22 J=1,N	CCP 4880
DO 22 I=1,N	CCP 4890
22 Q(I,J)=Q(I,J)/A(J,J)	CCP 4900
DO 23 I=1,N	CCP 4910
DO 23 J=1,N	CCP 4920
A(I,J)=0.0	CCP 4930
DO 23 L=1,N	CCP 4940
23 A(I,J)=A(I,J)+Q(I,L)*P(L,J)	CCP 4950
GO TO 25	CCP 4960
24 ISING=1	CCP 4980
25 RETURN	CCP 4990
END	

```

SUBROUTINE STPRINT (NSYS)
COMMON/FNAME/KE,LF(700),LS(700)/VALUES/V(700)
DO 10 I=1,NSYS
  II=NSYS+I
  10 PRINT 100,LF(I),V(I),LF(II),V(II)
  100 FORMAT(1X,A4,3H = E16.8,11X,A4,3H = E16.8)
END

```



```

SUBROUTINE EVALINT (MX,NX)
COMMON/FNAME/KF,LNAME(700),LSTART(700)/VALUE/V(700)
COMMON/COMP/KC,LCOM(4095),LCVM(4095)
COMMON/ICONS1/KCON,CT43(100)/LJ/LINFUN(10)
COMMON ISTORE(+095)
DIMENSION VBR(2),ZR(2)
EQUIVALENCE (VBR,VRR),(Z,ZR)
TYPE INT3 (2) VX,VL,VV,MES,TSTORE,V,CT44,AVAILINT,Z
TYPE INT3 (2) SINS,COS5,EXP5,LOG5,ATAN5
DATA (MH=7777777000000003),(LS=1000000003),(MSL=777)
100 J=NSCAN(KF,LNAME,NX)
    IF (J.EQ.0) 101,102
101 ZR(1)=0.
    ZR(2)=0.
    VX=Z
    RETURN
ENTRY EVALINT2
J=NX
102 M=NSTART(J)
103 NVL=LCVM(M),AND,MH
    NVR=LCVM(M)+LSH
    NVR=NVR+AND,MH
    VL=ANALINT(NVL,J2)
    IF (J2) 104,105
104 VR=ANALINT(NVR,J2)
    IF (J2) 109,105
105 CALL PRINT1 (NX)
    STOP
109 NUP=LCOM(M),AND,MH
    K=NSCAN(10,LJFUN,NOP)
    GO TO (110,120,130,140,150,160,170,180,190,195),K
110 RES=VL+VR
    GO TO 200
120 RES=VL-VR
    GO TO 200
130 RES=VL*VR
    GO TO 200
140 RES=VI/VR

```

GO TO 200	
150 IF(J2.(E.9)151.152	
151 KVR=(VRH(1)+.0001+VRH(2)+.0001)/2.	
RES=VL**KVR	
GO TO 200	
152 RES=VI**VR	
GO TO 200	
160 RES=STN5(VR)	
GO TO 200	
170 RES=COSS(VR)	
GO TO 200	
180 RES=EXP5(VR)	
GO TO 200	
190 RES=LOG5(VR)	
GO TO 200	
195 RES=ATAN5(VR)	
200 KDES=LCOM(M).AND.MSL	
IF(KDES.EQ.600)250.201	
201 KD=LCOM(M)/44	
KD=KD.AND.7777R	
TSTORE(KD)=RES	
M=M+1	
GO TO 103	
250 V(J)=VX=RES	
Z=VX	
END	

```

FUNCTION ANALINT (NV,J)
COMMON/ENAME/KF,LNAME(700),LSTART(700) /IVALUE/V(700)
COMMON TSTORE(4095)
COMMON/ICONST/KCON,CTAB(100)
DIMENSION ZZ(2)
EQUIVALENCE (Z,ZZ)
TYPE INT5 (2) V,CTAB,TSTORE,ANALINT,Z
KI=NV,AND,770000000000B
IF (KI.EQ.600000000000B)110,101
101 J=NV/100000000000B
IF (KI.EQ. 630000000000B)120,102
102 IF (KI.EQ. 230000000000B)130,103
103 PRINT 104
104 FORMAT(26H0ILLEGAL VARIABLE DETECTED )
J=0
RETURN
C
C F-VARIABLE
110 M=NSCAN(KF,LNAME,NV)
IF (M.EQ.0)111,112
111 ZZ(1)=0.
ZZ(2)=3.
ANALINT=Z
GO TO 121
112 ANALINT=V(M)
GO TO 121
C
C T-VARIABLE
120 ANALINT=TSTORE(J)
121 J=10
122 RETURN
C
C CONSTANT
130 ANALINT=CTAB(J)
END

```

**BLANK PAGE**

## EXPERIENCE WITH FORMAC AT HARRY DIAMOND LABORATORIES

David S. Marsh  
Harry Diamond Laboratories  
Washington, D. C.

[**ABSTRACT.** FORMAC is an experimental language and compiler, written by IBM, which allows the manipulation of algebraic symbols in much the same way that FORTRAN manipulates numerical values. It incorporates such FORTRAN features as subscripting and the DO loop capability. FORTRAN statements can be included in a FORMAC program so that results can be derived symbolically and evaluated numerically in the same program. FORMAC is particularly useful in those long, tedious algebraic problems which are so subject to copying and other errors when done with pencil and paper.]

The paper describes several small practice problems with which programmers became familiar with the language, its operation, and some of the commands. One larger problem is included, that of forming the determinant of a matrix, the elements of which are algebraic expressions.]

FORMAC (FOrmular MANipulation Compiler) is a combination of a compiler and a language which makes possible the manipulation of algebraic symbols as symbols, according to the rules of algebra, in the computer. FORTRAN, in comparison, performs in much the same manner with numbers.

FORMAC was written by IBM's Boston Advanced Programming department at Cambridge, Massachusetts. It is still an experimental system and was released unofficially for tests under actual operating conditions and to find out just what capabilities the computing community thought such a system should have.

Actually, FORMAC for the IBM 7090/7094 is no longer being developed by IBM since they are working on software (including an improved FORMAC) for the 360 series computers. Under the auspices of SHARE, however, a group at Wright-Patterson Air Force Base is taking over the further development of this system<sup>1</sup>.

FORMAC has been available at Harry Diamond Laboratories since early in 1966. Some of our uses of it will be presented here.

---

<sup>1</sup>Proceedings of SHARE XXVIII, p 4-93.

This paper is not a detailed tutorial discourse on FORMAC. It is, rather, a brief description, with some simple examples, of the language and its use. Hopefully, with this information you may be able to judge for yourselves whether FORMAC would be of value to you in your own operations.

FORMAC was written as an addition to and extension of FORTRAN. FORTRAN and FORMAC statements may be intermixed in a program. A FORMAC program goes through a pre-processor which translates FORMAC statements into FORTRAN "CALL" statements. The program then goes to the FORTRAN compiler. During execution as a FORTRAN program, the former FORMAC statements call special subroutines (added to the FORTRAN library) to accomplish their purposes.

	LET
+	SUBST
-	EXPAND
*	COEFF
/	PART
**	ORDER
FMCEXP	EVAL
FMCLG	FIND
	MATCH
FMCSIN	CENSUS
FMCCOS	BCDCON
FMCATN	ALGCON
FMCHTN	ERASE
	AUTSIM
FMCFAC	FMCDMP
FMCDFC	ATOMIC
FMCOMB	DEPEND
	PARAM
FMCDIF	SYMARG

Figure 1. FORMAC Commands

Figure 1 shows a list of available commands which gives a fair idea of FORMAC's capabilities. There are fifteen operators, which perform the purely mathematical functions, and nineteen declarative and executable statements used to define terms at the beginning of the program, manipulation expressions in various ways, and for various "housekeeping" purposes during the run. The mathematical operators are largely self-explanatory; most have direct FORTRAN counterparts. Among those which don't are FMCFAC and FMCDFC which perform the factorial and

double factorial functions, respectively. Similarly, FMCOMB performs the combinatorial function. FMCDIF performs differentiation.

The use of many of the declarative and executable statements is illustrated in program listings later in the paper.

Two obvious omissions from the mathematical operator list are commands for integrating and factoring. There exist no general algorithms for these processes.

In general, FORMAC seems best suited to performing relatively simple mathematical operations on relatively large and complicated algebraic expressions. The sample problems will illustrate this and show how some of the commands are used.

The first two problems are the generation and differentiation of the Lagrange interpolation formula. During the application of the Method of Steep Descent, it is desirable to find the value of X corresponding to the minimum point on a parabola passed through three known points. Given three points, the Lagrange formula (Fig. 2) yields the value of Y, lying on the parabola which passes through the known points, for any value of X. Differentiating the formula, setting the results equal to zero, and solving for X gives an expression which locates, in X, the minimum point of the parabola.

$$Y = Y_1 \left[ \frac{(X-X_2)(X-X_3)}{(X_1-X_2)(X_1-X_3)} \right] + Y_2 \left[ \frac{(X-X_1)(X-X_3)}{(X_2-X_1)(X_2-X_3)} \right] + Y_3 \left[ \frac{(X-X_1)(X-X_2)}{(X_3-X_2)(X_3-X_1)} \right]$$

Figure 2. Lagrange Interpolation Formula

The pattern of the subscripts in the formula suggests the operation of two nested DO loops. The inner loop would manipulate the subscripts within a term while the outer loop would multiply in an appropriately subscripted Y and sum up the expression. These loops formed the basis of the program to generate the formula. (See Figure 3.)

Figure 3 also shows intermediate results at the end of the first and second executions of the inner loop, the first execution of the outer loop, and the complete expression at the end of the third and last execution of the outer loop. These results show XX for the subscripted X of the formula and W for the subscripted Y terms. Figure 4 shows the final form of the results with X substituted for XX and Y for W. It also shows the effect of a special output subroutine, supplied by IBM, which yields a format closer to normal algebra than the standard FORMAC output.



$$Y = AX^2 + BX + C$$

$$\frac{dY}{dX} = 2AX + B = 0$$

$$X = -\frac{B}{2A}$$

Figure 5. Operations Performed on Lagrange Formula

The second problem is to perform on the generated Lagrange formula the operations shown in Figure 5. Figure 6 shows the program, and Figure 7 shows the results before and after substituting Y for W and X for XX. Notice that the division of the coefficients in the answer is only implied by enclosing the denominator in parentheses and raising it to the -1 power.

The third problem is an example of the type of simple mathematics mentioned earlier: forming the determinant of a 3x3 matrix<sup>2</sup>. Figure 8 shows the elements of the matrix; each is an algebraic expression. Not only that, but each of the 42 underlined terms represents another algebraic expression which must be substituted. Before the substitution expressions are put into the matrix elements, however, there are substitutions to be made among themselves.

The sequence of operations to be accomplished is:

1. Make the substitutions among the substitution terms.
2. Put the new substitution terms into the matrix elements.
3. Set B equal to zero, a condition of the original problem.
4. Form the determinant.

This is exactly the type of "dog-work" which is so subject to error and thus so frustrating when done by hand. If N people do such a job, with a requirement for accurate final results, there are usually at least N different results to be reconciled. This is also just the type of problem for which FORMAC was created.

Figure 9 shows the factors before and after their internal substitutions. Only eight of the original nine terms are of further interest since the F1 term appears only in the others and not in the matrix elements, but the remaining eight are larger.

---

<sup>2</sup>Generation of the matrix is described in HDL TR-1316, "An Equation for Phase Velocities in a Partially Ionized Gas", H. D. Curchack and F. T. Harris, Harry Diamond Laboratories, Washington, D. C. 20438.

Figure 10 shows the matrix elements after the substitution of the enlarged factors. Where they could originally be printed on 19 lines, they now cover 99 lines. Three of the matrix elements go to zero when B is set to zero (Figure 11) and they are so located in the matrix (Figure 12) that a fourth element  $A_{32}$

$$\begin{array}{ccc} A_{11} & A_{12}=0 & A_{13} \\ A_{21}=0 & A_{22} & A_{23}=0 \\ A_{31} & A_{32} & A_{33} \end{array}$$

Figure 12. Location of Zero Elements

is eliminated from the determinant. The determinant is now the difference between the products along the two major diagonals (Figure 13). It is unquestionably messy, but cleaning it up by hand is certainly far easier than obtaining it from the original matrix by hand.

This has been a brief description of some of the capabilities of FORMAC. In areas for which it is suitable, it can be a very useful tool.

```

INPUT TO FORMAC PREPROCESSOR
$10FMC MAIN  NODECK
SYNARG
FMCDDP LATER
ATOMIC X,XX(3),X1,X2,X3,Y1,Y2,Y3
1  FORMAT(1H1,35HLAGRANGE INTERPOLATION FOR PARABOLA //)
   WRITE(6,1)
   LET YY=0
   DO 15 I=1,3
     LET VX=1.
     DO 11 J=1,3
       IF(J.EQ.1) GO TO 11
       LET YX=YX*(X-XX(J))/(XX(1)-XX(J))
11  CONTINUE
       LET YY=YY+VX*YX
15  CONTINUE
       FMCDDP NOW,CURRENT,(VY)
       LET Y=SUBST VY,(XX(1),X1),(XX(2),X2),(XX(3),X3),(W(1),V1),(W(2),
       Y2),(W(3),Y3)
       LET Y=ORDER Y,INC,FUL
       FMCDDP NOW,CURRENT,(Y)
       CALL EDIT(Y,120)
       STOP
       END

EXPRESSION
(X-XX(2))*(X-XX(3))*(XX(1)-XX(2))*(-1.0)*(XX(1)-XX(3))*(-1.0)$

EXPRESSION
W(1)*(X-XX(2))*(X-XX(3))*(XX(1)-XX(2))*(-1.0)*(XX(1)-XX(3))
+(-1.0)*W(2)*(X-XX(1))*(X-XX(3))*(-XX(1)+XX(2))*(-1.0)*(XX
(2)-XX(3))*(-1.0)*W(3)*(X-XX(1))*(X-XX(2))*(-XX(1)+XX(3))*
(-1.0)*(-XX(2)+XX(3))*(-1.0)$

EXPRESSION
W(1)*(X-XX(2))*(X-XX(3))*(XX(1)-XX(2))*(-1.0)*(XX(1)-XX(3))
+(-1.0)*W(2)*(X-XX(1))*(X-XX(3))*(-XX(1)+XX(2))*(-1.0)*(XX
(2)-XX(3))*(-1.0)*W(3)*(X-XX(1))*(X-XX(2))*(-XX(1)+XX(3))*
(-1.0)*(-XX(2)+XX(3))*(-1.0)$

```

Figure 3. FORMAC Program for Lagrange Interpolation Formula

$$\begin{aligned}
 & \frac{(X - X_1)(X - X_2)(X - X_3) - 1.0}{(X - X_1)(X - X_2)(X - X_3) - 1.0} \cdot Y_3 + (X - X_1)(X - X_2)(X - X_3) - 1.0 \\
 & \frac{1.0}{(X_2 - X_3) - 1.0} \cdot Y_2 + (X - X_2)(X - X_3)(X_1 - X_2) - 1.0 \cdot Y_1
 \end{aligned}$$

Figure 4. FORMAC Version of Lagrange Interpolation Formula

```

INPUT TO FORMAC PREPROCESSOR
$1BFMC MAIN  NODECK
SYMPAR
ATOMIC X,XX(3),M(3),X1,X2,X3,Y1,Y2,Y3
DIMENSION MBUF(20)
FMCDMP LATER
1  FORMAT(1H1,68HLAGRANGE INTERPOLATION FOR PARABOLA AND DIFFERENTIAT
5  FORMAT(//1X,5HXMIN=,20A6 )
6  FORMAT(10F10.3)
7  FORMAT(//1X,6F10.5,1PE15.7)
8  FORMAT(//1X,20A6)
LET YY=M(1)+(X-XX(1))*(X-XX(2))/(X-XX(3))+(X-XX(1))*(X-XX(2))*(X-XX(1)-XX(3))
1  +M(2)*(X-XX(1))*(X-XX(3))/(X-XX(2))*(X-XX(1)-XX(3))
2  +M(3)*(X-XX(1))*(X-XX(2))/(X-XX(3))*(X-XX(1)-XX(2))
WRITE(6,1)
LET Y=EXPAND YY
LET DYDX=FMCDIF(Y,X,1)
LET CX1=COEFF DYDX,X,A
LET CX0=COEFF DYDX,X=0,B
LET ANS=CX0/CX1
LET BNS=ANS
LET BNS=ORDER BNS,INC,FUL
Q=0.
LET Q=BCDCON BNS,MBUF,20
WRITE(6,5)(MBUF(J),J=2,20)
14 IF(Q.EQ.0.) GO TO 13
LET Q=BCDCON BNS,MBUF,20
WRITE(6,8)(MBUF(J),J=2,20)
GO TO 14
13 CONTINUE
LET Y=SUBST ANS,(XX(1),X1),(XX(2),X2),(XX(3),X3),(M(1),Y1),(M(2),
1 Y2),(M(3),Y3)
LET BAS=Y
LET BNS=ORDER BNS,INC,FUL
Q=0.
LET Q=BCDCON BNS,MBUF,20
WRITE(6,5)(MBUF(J),J=2,20)
15 IF(Q.EQ.0.) GO TO 16
LET Q=BCDCON BNS,MBUF,20
WRITE(6,8)(MBUF(J),J=2,20)
GO TO 15
16 CONTINUE
AAA PARAM (X1,1),(X2,3),(X3,6),(Y1,-7),(Y2,5),(Y3,14)
LET ANS=EVAL Y,AAA
WRITE(6,7)X1,X2,X3,Y1,Y2,Y3,ANS
STOP
END

```

Figure 6. FORMAC Program to Differentiate Lagrange Formula



```

LET A(1,1)=-FV*(SQOMP+FC*(1.+F+FE))
-B*FC*OY*OY*(1.+NNA*FI)$

LET A(1,2)=B*FC*OX*OY*(1.+NNA*FI)$

LET A(1,3)=OY*FC*FV$

LET A(2,1)=-FP*8*8*OX*OY*(1.+NNA*FI)*(1.+4.*8*F+2.*FI)$

LET A(2,2)=B*FP*(-X*X*(1.+6./5.*(FI+F)+NNA*FI+1.2*(1.+NNA)*FI*FE)
+EV*(FI+2.*FI)*(1.-SQOMP+F+FE))
-X*X*FI1*((1.+2.*8*F+2.*FI)*FV-X*X*(1.+NNA*FI+1.2*FI))
-X*X*F21*((1.+2.*8*F+2.*(1.+NNA)*FI)*FV-X*X*(1.+1.2*(1.+NNA)*FI))
+8*8*OX*OX*FP*(FI+2.*FI)*(1.+NNA*FI)$

LET A(2,3)=8*OX*(FI+2.*FI)*EV*FP
-8*OX*X*X*(FI)*FV*(1.+2.*8*F+2.*FI)-X*X*(1.+NNA*FI+1.2*FI)
-F20*2.*8*FI*(FV*(1.+2.*8*F+2.*(1.+NNA)*FI)-X*X*(1.+1.2*(1.+NNA)
*FI))+8*FP*(1.+NNA*FI+1.2*FI)$

LET A(3,1)=-FC*FP*FV*OY$

LET A(3,2)=FC*FP*FV*OX+FC*OX*X*(FI+21*FI+NNA*FI)$

LET A(3,3)=-FP*FV*(SQOMP+FC*(1.+F+FE))
-(OX*OX*OY*OY)*B*FC*FP*(1.+NNA*FI)
+OX*OX*B*FC*X*X*(F10-F20*2.*FI+NNA*FI*F10)$

```

Figure 8. Matrix Elements



$$\begin{aligned}
 & \text{3 FCG} \quad -C \times X \times 2.0 - 1.0 \$ \\
 & \text{5 FUG} \quad -FI \times (-FI + 2.0 \times FV) \times 2.0 \times (FI + (6.00000001E-1 \times FI \times (2.0 \times NNA + 1.0) \times 1.0) \times FT) \times X \times 2.0 \$ \\
 & \text{6 FUG} \quad -(-2.399999998 \times B \times (F + FE) \times FI + (1.19999999 \times FI \times (NNA + 1.0) \times 1.0) \times FT) \times X \times 2.0 \times FI \times FV \$ \\
 & \text{9 FPG} \quad -((2.0 \times B \times (F + FE) \times FI \times (2.0 \times NNA + 1.0) \times 1.0) \times FI + (2.0 \times NNA + 1.0) \times 2.0) \times FI + (1.19999999 \times FI \times (NNA + 1.0) \times (2.0 \times FI \times NNA + 1.0) \times (FI \times (2.0 \times NNA + 1.0) \times 2.0) \times 1.0) \times FT) \times X \times 2.0 \times FI \times FV \times (FI + 2.0 \times (1.19999999 \times FI \times (2.0 \times NNA + 1.0) \times 1.0) \times FT) \times X \times 4.0 \$ \\
 & \text{4 F1G} \quad -B \times (-1.599999998 \times (F + FE) \times FI \times FV + (-FE \times FI) \times (2.0 \times FI \times (NNA + 1.0) \times 1.0) \times 1.799999998) \times 2.0 \times (-1.19999999 \times B \times (F + FE) \times (-FI + 2.0 \times FV) + 1.0E-1 \times FT - FV) \times FI \times X \times 2.0 \$ \\
 & \text{2 PV G} \quad -FI \times (NNA + 1.0) \times 1.0 \$ \\
 & \text{7 F1UG} \quad -2.0 \times B \times (F + FE) \times FI - 4.0 \times B \times FE \times FV + (-FI \times (NNA + 1.0) - 8.999999998E-1) \times FT \$ \\
 & \text{1 FT G} \quad -4.0 \times B \times F + 1.0 \$ \\
 & \text{4 F1G} \quad -2.0 \times (2.0 \times B \times (F + FE) + 1.0) \times FI + (2.0 \times FI \times NNA + 1.0) \times FT \$ \\
 & \text{3 GSUBST=C \times X \times 2.0 - 1.0 \$} \\
 & \text{5 GSUBST=-(4.0 \times B \times F + 1.0) \times (2.0 \times FI \times NNA + 1.0) \times 2.0 \times (2.0 \times B \times (F + FE) + 1.0) \times FI) \times (-FI + 2.0 \times (FI \times (NNA + 1.0) \times 1.0) \times 2.0 \times ((4.0 \times B \times F + 1.0) \times (6.00000001E-1 \times FI \times (2.0 \times NNA + 1.0) \times 1.0) \times FI) \times X \times 2.0 \$} \\
 & \text{6 GSUBST=-(2.399999998 \times B \times (F + FE) \times FI + (4.0 \times B \times F + 1.0) \times (1.19999999 \times FI \times (NNA + 1.0) \times 1.0) \times X \times 2.0 \times ((4.0 \times B \times F + 1.0) \times (2.0 \times FI \times NNA + 1.0) \times 2.0 \times (2.0 \times B \times (F + FE) + 1.0) \times FI) \times (FI \times (NNA + 1.0) \times 1.0) \$} \\
 & \text{9 GSUBST=((4.0 \times B \times F + 1.0) \times (2.0 \times FI \times NNA + 1.0) \times 2.0 \times (2.0 \times B \times (F + FE) + 1.0) \times FI) \times (FI \times (NNA + 1.0) \times 1.0) \times (-((4.0 \times B \times F + 1.0) \times (1.19999999 \times FI \times (NNA + 1.0) \times (2.0 \times FI \times NNA + 1.0) \times (FI \times (2.0 \times NNA + 1.0) \times 1.0) \times 2.0) \times FI) \times (FI \times (2.0 \times NNA + 1.0) \times 1.0) \times 2.0) \times FI \times (2.0 \times NNA + 1.0) \times 1.0) \times FI) \times X \times 4.0 \$} \\
 & \text{4 GSUBST=B \times (-1.599999998 \times (F + FE) \times FI \times (FI \times (NNA + 1.0) \times 1.0) \times (-FE \times FI) \times (2.0 \times FI \times (NNA + 1.0) \times 1.799999998) \times 2.0 \times (-1.19999999 \times B \times (F + FE) \times (-FI + 2.0 \times (FI \times (NNA + 1.0) \times 1.0) \times 1.0E-1 \times (4.0 \times B \times F + 1.0) - FI \times (NNA + 1.0) - 1.0) \times FI \times X \times 2.0 \$} \\
 & \text{2 GSUBST=FI \times (NNA + 1.0) \times 1.0 \$} \\
 & \text{7 GSUBST=2.0 \times B \times (F + FE) \times FI - 4.0 \times B \times FE \times (FI \times (NNA + 1.0) \times 1.0) \times (4.0 \times B \times F + 1.0) \times (-FI \times (NNA + 1.0) - 8.999999998E-1) \$} \\
 & \text{1 GSUBST=4.0 \times B \times F + 1.0 \$} \\
 & \text{4 GSUBST=(4.0 \times B \times F + 1.0) \times (2.0 \times FI \times NNA + 1.0) \times 2.0 \times (2.0 \times B \times (F + FE) + 1.0) \times FI \$}
 \end{aligned}$$

Figure 9. Substitution Terms Before and After Internal Substitutions

[illegible]

72





## A SIMPLE ELECTRONIC TRUE RANDOM EVENT GENERATOR

D.R. Koehler, J.T. Grissom, and R.G. Polk  
U.S. Army Missile Command, Redstone Arsenal, Alabama

**ABSTRACT.** A device is proposed which will generate a uniform series of random binary digits. This device could be considered an electronic equivalent of a coin-flipping machine in that its output is a continuous series of binary digits with successive digits having exactly equal probabilities of being "1" or "0". Such a device would be ideally suited to the on-line production of random numbers for use in Monte Carlo calculations by digital computers. With suitable combinatorial logic, generation of random pulses or random analog signals could easily be accomplished. The device as presently conceived is small, compact, uncritical, and requires little power. Using the space-randomness of particle emission from a radioactive source and two small semi-conductor detectors as a signal generator, plus a few readily available integrated micro-circuit packages, the device could be packaged on a medium-sized circuit board. Interfacing to any of the present generation of digital or hybrid computers would present no problems, and the bit generation rate could be adjusted to satisfy the demand rate of the fastest of today's computers.

Computer technology presently has reached such a state of development that today computer systems are being built which are so large that seemingly the necessary software and programs to utilize them cannot be produced. The burgeoning field of computer systems application is working overtime searching for ways and means to fully occupy the vast capabilities of the very large computer systems, and problems which seemed impossible of solution by any computer technique a few years ago are beginning to yield to new approaches made possible by these large new machines. In particular, one long-popular but computationally expensive numerical technique known as the "Monte Carlo calculation" is seeing a period of rapid development as a line of attack on problems which would not yield to ordinary analytical and numerical techniques. The long-standing problem with most Monte Carlo programs is their requirement for random numbers in large quantities.

Computer-users in the areas of statistical sampling and simulation, Monte Carlo calculations, and the promising new field of "stochastic" computation so far have been steadily handicapped by the difficulty of obtaining high-quality random numbers for their programs. In particular, the stochastic computer requires numbers in great quantity and of high quality, and speed of computation is directly dependent on the rate at which random numbers can be provided to the computer. Computer designers so far seem to have virtually ignored this problem altogether, leaving it up to the programmers to somehow devise a technique of getting numbers.

The common techniques, up to this time, have been the insertion of tables of random numbers in the computer memory, or the calculation of "pseudo-random" numbers arithmetically via a short in-computer program using any one of quite a number of possible algorithms. Both of these approaches suffer from requiring

memory space, and both are limited in the quantity and quality of numbers which can be supplied. Furthermore, algorithmic solutions require non-negligible amounts of computer time. The real solution to the problem will come when a good random number generator can be built which will produce all manner of random numbers any program or computer may require and which can be hooked up to the computer directly.

Attempts have been made to construct random number devices, and their history makes interesting reading. But the end product of most of these attempts seems generally to have been slow in speed, cumbersome, unwieldy, and unsuited for direct connection to the computer; or else complicated, sophisticated, lacking stability, and requiring much careful adjustment and attention. We shall not take time to discuss any of these devices here. The interested reader will find references on some of these devices in the bibliography.

We propose, as have many others interested in this problem, a device based upon the random nature of the decay of radioactive substances. However, instead of mixing radioactivity detectors with clock-pulse generators and observing the time-randomness of emission of nuclear particles, as has been the traditional approach, we would like to combine two reasonably identical and independent nuclear detector systems whose average count rates are exactly equal. The time-and-space randomness of the decay of the radionuclide then requires that at any given instant of time there be exactly equal probabilities that either detector will receive the next particle. If one detector were labeled "heads" and the other "tails", the output pulses of the two detectors would be just as good for decision making as the ubiquitous coin, and much, much faster.

The proposed device is shown schematically in Figure 1. The "sandwich" of detectors and radioactive source can be made quite compact. It could be fitted on one corner of a single printed-circuit board, or even on a single chip of silicon which at the same time could carry some of the necessary active electronics. The source strength even for very high count rates could be relatively weak and quite harmless - less damaging than an ordinary radium watch dial. Using ordinary silicon semiconductor radiation detectors, the device could be made to pump out random binary bits at a rate fast enough even for the "stochastic" computers: and as computer technology advances, the permissible bit generation rate can advance with it, since virtually all the associated electronics can be digital and will benefit from improvements in digital techniques.

The "sandwich" of Figure 1 is not exactly a proper configuration for direct connection to any user device, such as a computer. First of all, the detector signals are small and must be amplified. Then some means must be incorporated to convert the amplified detector pulses to the necessary logic levels for feeding the user device. In Figure 2 we see a possible realization of a generator of serial binary bits. The "conversion" flip-flop is triggered by the detector pulses into "1" or "0" states and thus provides logic levels representing the two binary digits. These digits are produced one after the other in serial fashion by "inspecting" the logic



levels every time a detector pulse appears at the "clock" output and delivering to the user device the proper binary bit as determined by the state of the flip-flop.

For a random pulse generator, or some sort of special noise generator, this configuration might serve admirably. But a computer likes its input to be more regular, the time-randomness of binary output of this serial generator would be unacceptable to the computer systems designer. Therefore some sort of buffer memory must be incorporated. Possibly the easiest solution to this problem is the addition of a shift register which is driven by the outputs of the serial binary generator. This is shown in Figure 3. Any time the computer desired a new random number, it could sample the state of the shift register and transfer its contents via parallel-access lines to the processor, or else the transfer of digits from the generator to the register could be temporarily halted while the number contained in the register is clocked out at the computer clock rate and fed to the computer serially from the back of the shift register.

The ultimate choice of means of converting the detector "sandwich" pulses into numbers in the computer will be up to the computer designer. Our suggestions are only for illustrating the possibilities. For the sake of simplicity, we have so far ignored one very important additional element of the total generator. That element consists of the means by which the generator is stabilized so as to maintain the exactly equal count rates we presupposed as the necessary condition for true randomness. For certain types of radioactive sources and preamplifiers, this "stabilization" can be so simple as a micrometer adjustment of the position of the source between the detectors - the inherent counting stability of the remainder of the system will be high enough that over periods of perhaps a year or more between maintenance checks the drift and count rate inequality will be quite negligible.

Unfortunately, the type of source presupposed above could be rather "hot" as radioactive sources go, and might prove something of a problem around a computer. Also the adjustment mechanism would be somewhat bulky relative to the size of the rest of the system. A better approach probably would be the use of feedback stabilization. For example, in Figure 4 we have added an up-down scaler which continually measures the difference in the number of "1's" and the number of "0's," and if the difference exceeds a certain value, to be determined by statistical considerations, then an adjustment of the count rate in the one channel would be made via the second up-down scaler, DAC, and discriminator. This sort of stabilization scheme is basically digital, with a step-wise adjustment of the relative count rates, which should, after a stabilization period, lead to a steady-state condition in which the statistical probabilities of the two binary states fluctuate very slightly about the exact 50% level.

Having conceived the device, we naturally are curious as to just how good it might be. Unfortunately, it is not within our mission to do device development such as this, so we have not been able to obtain and patch-up the necessary logical elements to test it. However, some spare detectors,



amplifiers, and a paper tape punch were temporarily rigged to punch random bits in paper tape. The system had no provision for stabilization, and count rates were crudely adjusted to something near equality in both channels simply by adjusting channel gains. Something over  $10^5$  bits were punched out, which we converted to card and then gave to our Computation Center for testing. Considering the small sample we had to work with and the consequent rather large variance to be expected on any given test, no real conclusions could be developed as to the quality of the numbers. All results of all tests, however, were within statistical expectations based upon the known relative numbers of ones and zeros and otherwise assuming complete randomness.

## BIBLIOGRAPHY

This list of references is just a small "random" sampling of the large body of literature available on the subject of random numbers. Most of the articles listed give additional references, and several have quite extensive bibliographies covering both arithmetic generators and random number devices.

**EARLY DAYS.** Before computers and in the early days of computers, statisticians and mathematicians resorted to tables of digits compiled from hopefully uncorrelated batches of numbers. The first three references tell an interesting story of the days B. F. C. (Before Fast Computers).

1. Kendall and Smith, "Randomness and Random Sampling Numbers," Journal of the Royal Statistical Society **CI** (1938) 147-172. The classic treatise on tests for randomness. Herein were proposed for the first time the four basic tests - frequency, serial, poker, and gap - which for so many years were the foundation of random number testing.

2. H. B. Horton, "A Method for Obtaining Random Numbers," Annals of Mathematical Statistics **XIX** (1948) 81-85. Mr. Horton of the Interstate Commerce Commission compiled a small table of numbers from presumably uncorrelated railway freight car waybill numbers and subjected them to Kendall and Smith's elementary tests.

3. The Rand Corporation, A Million Random Digits with 100,000 Normal Deviates. The Free Press, Glencoe, Illinois. 1955. The table of digits is prefaced with a short dissertation on the "electronic roulette wheel" which the Rand group used to make up the table, as well as the tests they performed and the re-randomization technique they found necessary in order that the table test "random."

**ARITHMETIC COMPUTER METHODS.** Faster computers naturally attracted programmers more and more to the use of Monte Carlo techniques. The poor performance of random number devices led to the expenditure of considerable effort to devise ways of generating numbers in the computer itself.

4. Hull and Dobell, "Random Number Generators," SIAM Review **4** (1962) 230-254. An extensive discussion of the state of the art in 1962

of arithmetic generators, with some comments on random number devices. With extensive references and bibliography.

5. G. Marsaglia, "Random Variables and Computers," Boeing Scientific Research Laboratories Report Number D1-82-0182 (ASTIA 278,358), 1962. One of many, many treatises in the literature on arithmetic generators and linear transforms as component parts of the computer program. References.

6. Marsaglia and Bray, "A Small Procedure for Generating Normal Random Variables," Boeing Scientific Research Laboratories Report Number D1-82-0221 (ASTIA 294,455), 1962. Another example of the many papers published in this area. Here a method of converting a uniform distribution to a special one (the normal distribution) is presented.

7. McLaren and Marsaglia, "Uniform Random Number Generators," Journal ACM 12 (1965) 83-89. Here members of the Boeing group re-examine some of their numerical methods in the light of newer developments in testing, indicate some of their failings, and suggest some new techniques, including a return to the old standby, the random number tables.

8. R. P. Chambers, "Random-Number Generation on Digital Computers," IEEE Spectrum 4, No. 2 (Feb 1967) 48-56. Chambers discusses the old standby arithmetic methods plus some new ones and touches briefly on random number devices. Includes extensive references and bibliography.

9. Coveyou and MacPherson, "Fourier Analysis of Uniform Random Number Generators," Jour. ACM 14 (1967) 100-119. A new technique of testing is presented which points up the failure of many of the arithmetic techniques, particularly when used on small machines.

**RANDOM NUMBER DEVICES.** There is a considerable body of information on random number devices, but unfortunately much of it is buried in articles not specifically directed towards the treatment of devices and hardware. Many articles mention devices in passing, with the seeming implication that hardware has never progressed beyond the electronic "roulette wheel" of Kendall and Smith and the Rand group. The following three short papers are an indication that device development

has not been totally neglected, even though overall results to date may have been small.

10. F. Sterzer, "Random Number Generator Using Subharmonic Oscillators," Rev. Sci. Instr. 30 (1959) 241-243. Sterzer utilizes some microwave devices and techniques to obtain high production rates and reportedly better randomness than the Rand tables. The technology however likely will be a little foreign to the computer designer.

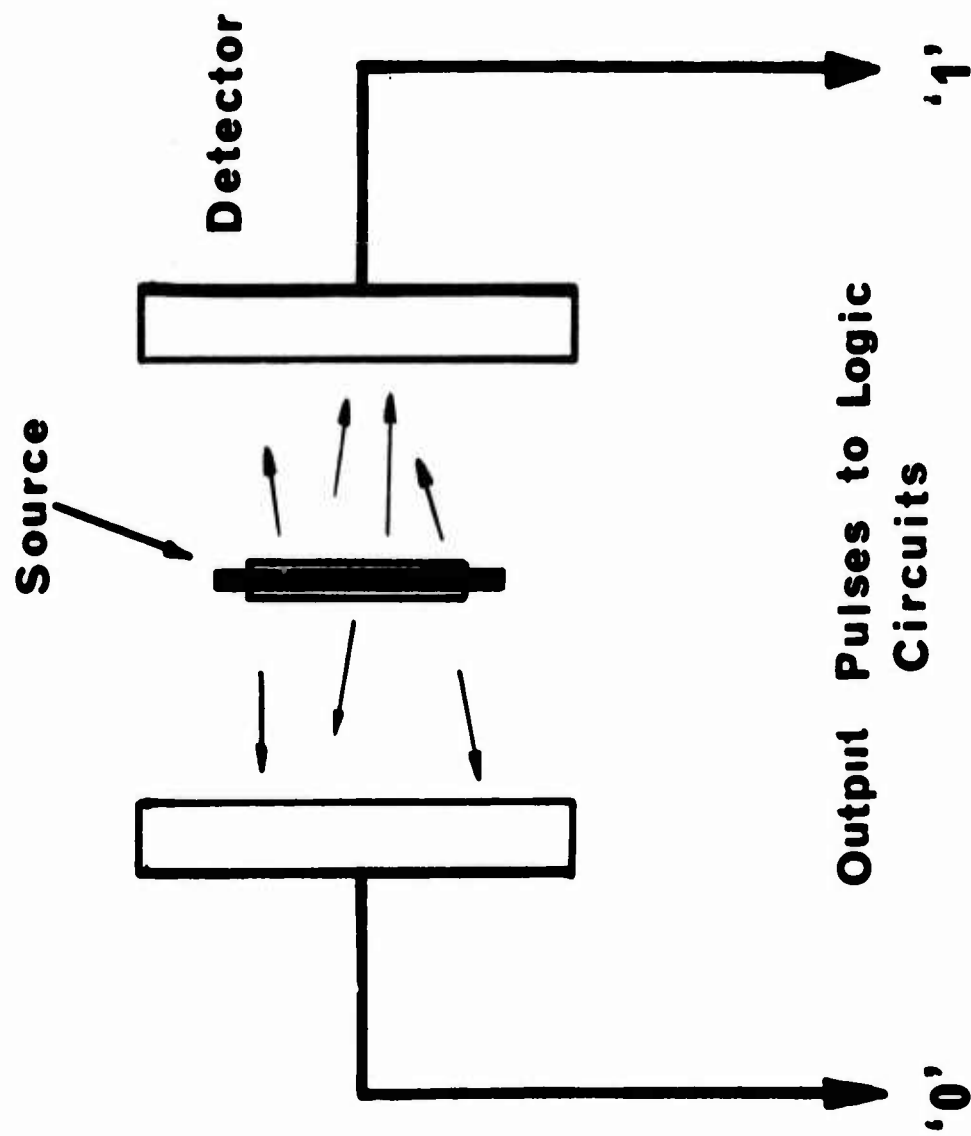
11. Dillard and Simmons, "An Electronic Generator of Random Numbers," IRE Transactions on Electronic Computers EC-11 (Apr 62) 284. A traditional approach using noisy thyratrons and considerable associated digital circuitry. Production rates of about 15-20 kbits per second.

12. Tait and Skinner, "A Random Signal Generator," Elec. Eng. 38 (1966) 2-7. A rather specialized device, but with an interesting method of obtaining random digits.

#### NEW VISTAS

13. B. R. Gaines, "Stochastic Computer Thrives on Noise," Electronics 40, No. 14 (10 Jul 67) 72-76. An introduction to the stochastic computer and its applications and promise for the future.

14. Koehler, Grissom, and Polk, A Random Pulse Generator. Patent Application #518,733, filed 4 Jan 66. The basic reference for the material of this paper.



**Fig.1. Basic Generator Sandwich**

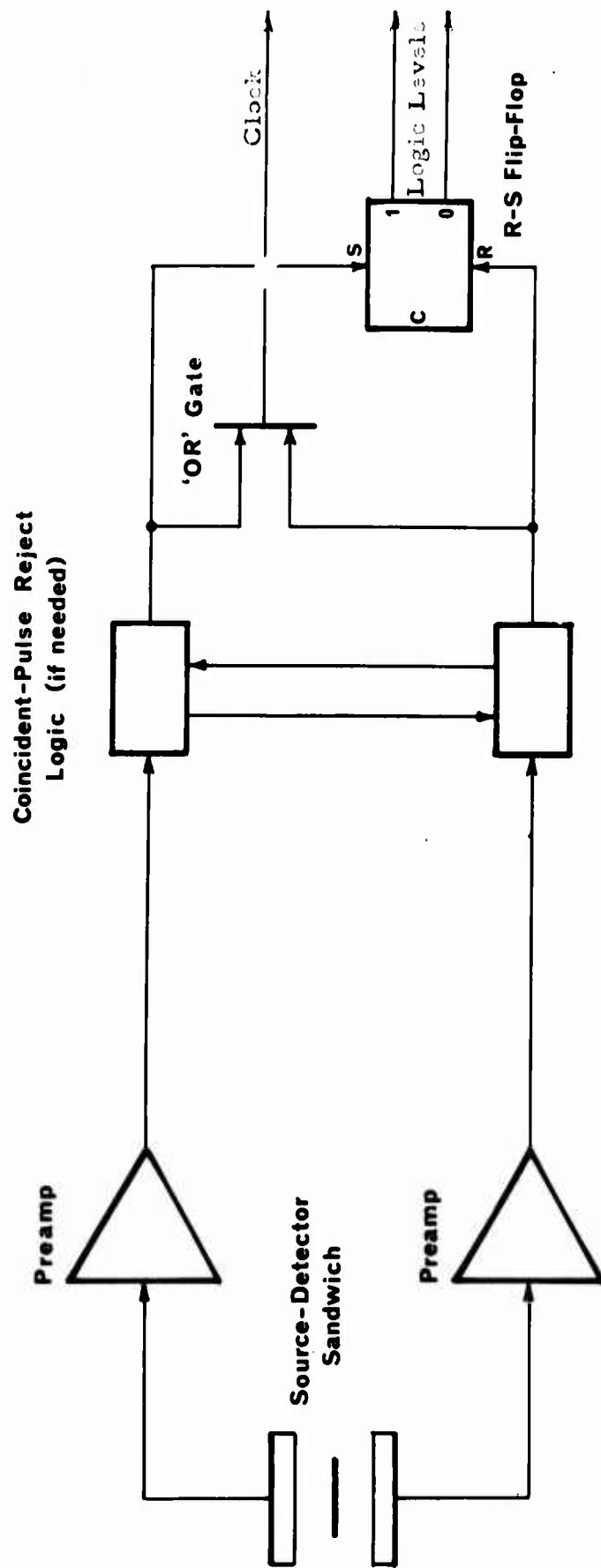


Fig. 2. Serial Binary Random Digit Generator

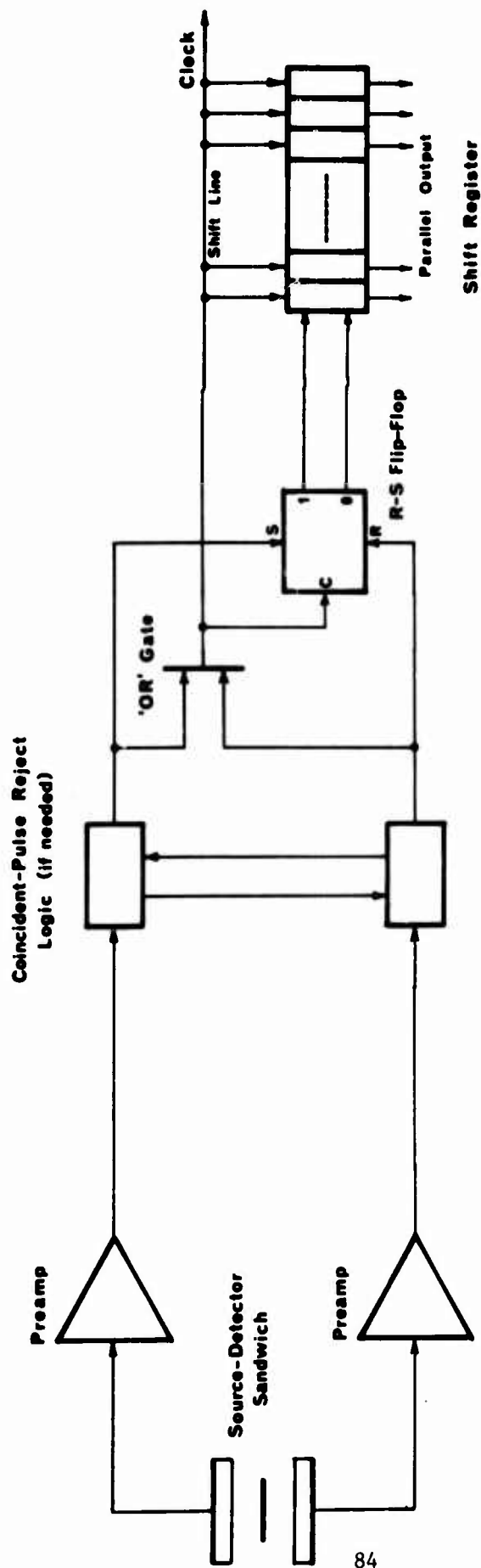


Fig. 3. Buffered Output Generator



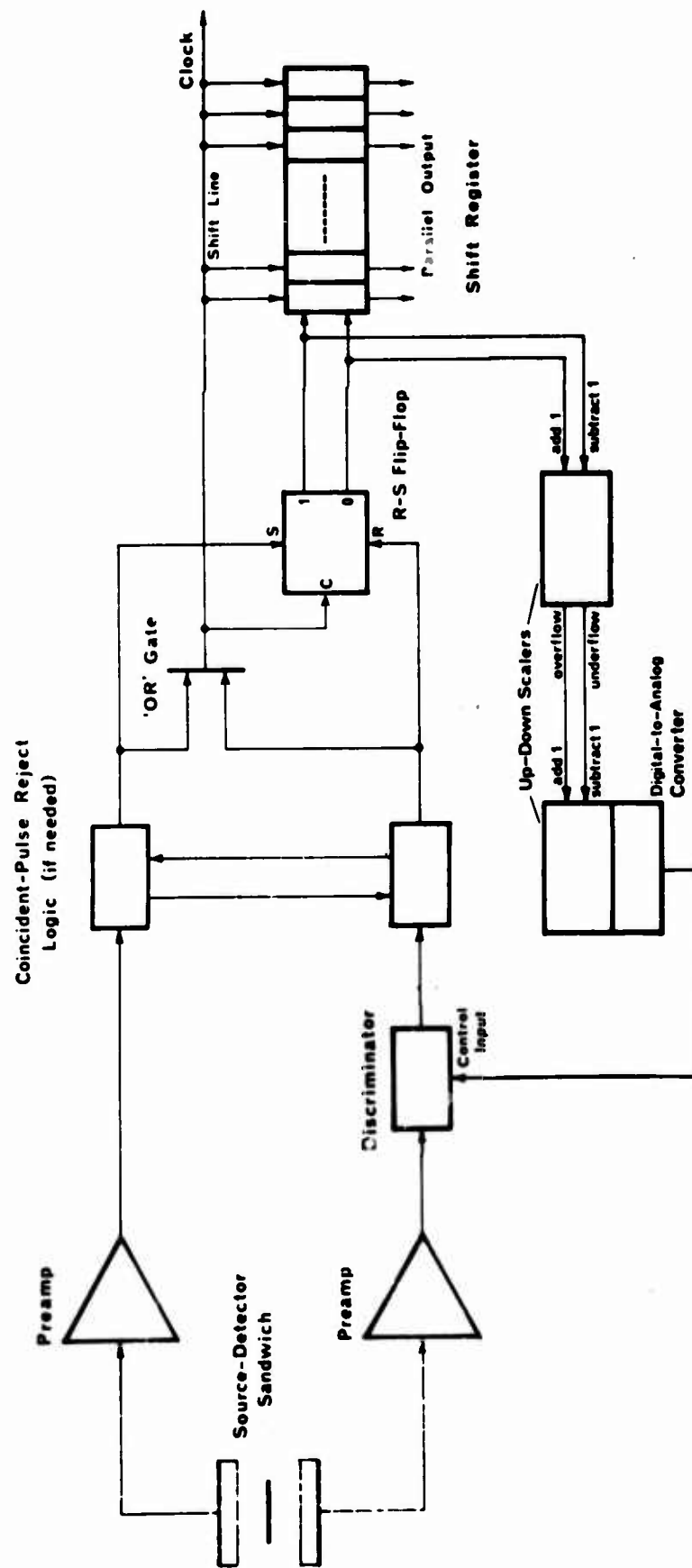


Fig. 4. Stabilized, Buffered Output Generator

**BLANK PAGE**

## PROGRAMMING INTERVAL ARITHMETIC AND APPLICATIONS

Allen Reiter  
Lockheed Missiles and Space Company  
Palo Alto, California

INTRODUCTION. This paper discusses the current state-of-the-art in interval arithmetic, both from the programming point of view and from the point of view of applications to date.

Interval arithmetic was first developed formally by R.E. Moore around 1960, although there is essentially nothing new in the concepts involved. Moore originally envisioned interval arithmetic as a means of completely rigorous automatic error control for computational processes using a digital computer. More recently, people have begun to appreciate the potential of interval arithmetic for control theory, and also as a tool in experimental designing on-line, with both a man and a computer as parts of the feedback loop.

There are basically three different sources of error associated with numerical computations. The first, which we may call the data problem, is due to the fact that the value of some given parameter may not be known exactly (this is for example true for physically-determined parameter values), or else may not be exactly represented in a computer (for example, the number  $\pi$ ). A second type of error, usually called truncation error, is caused by the necessity to terminate after a finite number of steps some infinite converging process, or (equivalently) by the requirement that some well-defined expression be evaluated at some point whose location is known only approximately (for example, the remainder term of the Taylor series with remainder). The third type of error is round-off error, caused by the necessity to restrict computational processes to operate on numbers which do not exceed some predetermined number of digits in length. Round-off error has traditionally been the most troublesome, primarily because of its non-analyticity. Attempts at rigorous "pencil-and-paper" bounding of round-off either are too difficult or lead to hopelessly pessimistic "bounds".

Interval arithmetic keeps track of the accumulation of error by continually producing an interval, guaranteed to contain the "true" result, and performing the indicated arithmetic operations on the entire interval. Since the implementation of interval arithmetic necessarily involves ordinary arithmetic operations on the end-points of the interval, which in turn involve rounding, care must be taken to perform the rounding properly: "down" for the left-hand end point, and "up" at the right-hand one. Thus, when in the sequel we shall speak of interval arithmetic, it shall be understood that in the implementation of the operations on a computer rounded interval arithmetic is used. However, in the formal discussion of interval arithmetic we shall ignore this fact, and define the formal operations independently of their implementation.

ARITHMETIC RULES. An interval is simply a closed interval on the real line, of the form  $[a,b]$ . We can also think of an interval as a fuzzy number

$x$  of the form  $[x-\epsilon, x+\epsilon]$ ; although  $\epsilon$  is certainly not restricted to being small in any sense. The arithmetic operations are defined in a natural fashion, and in fact reduce to ordinary arithmetic when  $\epsilon=0$ . (When the occasion arises, we shall speak of ordinary real numbers as degenerate intervals.)

Elementary operations are defined as follows. Let  $[a,b]$  and  $[c,d]$  be a pair of intervals. Then

$$\begin{aligned} [a,b] + [c,d] &= [a+c, b+d] ; \\ [a,b] - [c,d] &= [a-d, b-c] ; \\ [a,b] * [c,d] &= [\min(ac, ad, bc, bd), \max(ac, ad, bc, bd)] ; \\ [a,b] / [c,d] &= [a,b] * [1/d, 1/c] ; \text{ (division is defined} \\ &\text{only if the interval } [c,d] \text{ does not contain the point zero).} \end{aligned}$$

It can be seen that these operations are defined in such a way that the result is precisely the set of all possible values of the operation as the operands range over the argument intervals.

Interval arithmetic is associative, and addition and multiplication are commutative. Unfortunately, the distributive law does not hold; instead we have the "subdistributive" law ( $I, J$ , and  $K$  being intervals):

$$I * (J + K) \subset I * J + I * K.$$

That the inclusion can indeed be proper can be seen from the example

$$[-3,3] * [0,2] + [-3,3] * [-1,0] = [-6,6] + [-3,3] = [-9,9] ,$$

whereas

$$[-3,3] * ([0,2] + [-1,0]) = [-3,3] * [-1,2] = [-6,6] .$$

The example also illustrates that a given interval number may have many multiplicative units: if  $y$  is any real number in  $(-1,1)$ , then all interval numbers of the form  $[-1,y]$  or of the form  $[y,1]$  are multiplicative units for the interval number  $[-3,3]$ .

More disruptive is the fact that although an additive unit is unique ( $[0,0]$ ), interval numbers do not in general possess additive inverses. (This reflects the fact that once uncertainty or error has been introduced into a computational process, it cannot be cancelled out, but must be carried along till the end.) This last property is responsible for almost all of the difficulties in interval arithmetic, and frequently necessitates very delicate handling of the specification of a computational algorithm - something that the current state-of-the-art is not quite up to. (In spite of this handicap, useful areas of application have already been found.)

The usefulness of interval arithmetic for error bounding comes from the fact that

- 1) The elementary arithmetic operations are continuous mappings from  $I_1 \times I_2$  onto  $I_3$  (the  $I$ 's are arbitrary intervals);
- 2) Since the elementary operations are defined in such a manner that the range of the operator as the operands range over the argument intervals is contained in the result interval, the same is true for any well-defined grouping of such operations on argument intervals; in other words, for all rational functions. Of course, rational operations is all computers are capable of executing; thus, any computable function can be bounded by the use of interval arithmetic.

Let  $f(x_1, \dots, x_n)$  be a given formal rational function in the indeterminates  $x_1, \dots, x_n$ . When the indeterminates take on real values,  $f$  denotes a real-valued function. There may be many different ways of representing this function, which are all algebraically equivalent; we will fix a representation  $f_1(x_1, \dots, x_n)$ . If we let the indeterminates take on interval values  $X_1, \dots, X_n$ , then the function  $f_1$  is still well-defined (we can regard  $f_1$  as a computer program, with a sequence of arithmetic operations to be carried out in a certain order); we however choose to call this interval-valued function  $F_1(X_1, \dots, X_n)$ . Note that the fact that  $f_1$  and  $f_2$  may be algebraically equivalent to  $f$  (and to each other) certainly does not imply that  $F_1$  and  $F_2$  are equivalent (this is primarily due to the failure of the cancellation law for interval arithmetic). The basic theorem of interval arithmetic however states that for the purposes of error bounding any representation will do:

Theorem. Let  $f$  be a given rational function,  $f = f(x_1, \dots, x_n)$ , and let  $F$  be any representation of  $f$ ,  $F$  to be evaluated in interval arithmetic. Let  $X_1, \dots, X_n$  be a collection of closed intervals on the real line. Then the range of  $f$  as each variable  $x_i$  ranges over  $X_i$  is contained in  $F(X_1, \dots, X_n)$ .

The theorem assures us that interval arithmetic is sufficient to compute bounds on the range of a rational function over a compact rectangle in  $E_n$ . Note that since the evaluation of  $F$  can be done using rounded interval arithmetic, the round-off error is included in the final bounds produced by  $F$ . (It is worth while stressing though that nothing is said about bounding the round-off that might occur in evaluating  $f$ . The round-off process is not a continuous operation. On some computers, in particular on the IBM SYSTEM/360, it is easy to cook up examples where  $f$  evaluated at some point  $p$  inside the rectangle turns out to be outside the interval obtained by evaluating  $F$ . This is but another aspect of "dirty" floating-point hardware. The true range of  $f$  is however always contained in  $F$ .)

As already noted, the width of the interval obtained by evaluating

F may be considerably greater than the width of the true range of f; it is also generally quite sensitive to the choice for the particular representation F. This shall be discussed below.

SOME APPLICATIONS OF INTERVAL ARITHMETIC. Aside from the obvious advantage of providing error bounds, interval arithmetic can be used by a computer to control the growth of error. While potentially the realm of applications is unlimited, the author knows only of the following contexts in which interval arithmetic has been studied:

- a) The initial-value problem for ordinary differential equations;
- b) Finding roots of polynomials;
- c) Matrix inversion, and the eigen-value problem for matrices;
- d) Solution of systems of simultaneous (non-linear) equations;
- e) The two-point boundary-value problem.

In these areas, analytic techniques are being developed which make use of interval arithmetic evaluations, and which also address themselves to the peculiar problems which arise in using interval arithmetic.

THE INITIAL VALUE PROBLEM FOR ORDINARY DIFFERENTIAL EQUATIONS. Let  $dy/dx = f(x,y)$  denote a system of n first-order ordinary differential equations, and let  $y_0 = y(x_0)$  be given. The application of interval arithmetic to the automatic generation of solutions to this problem was the first application suggested by Moore. He designed a computer program using interval arithmetic which gave solutions with automatic error bounds.

His method is described in [7]. Briefly stated, the solution is expanded in a Taylor series with remainder (up to a specified number of terms) at a given point. To bound the remainder term, the required derivative is evaluated over a whole rectangle (using interval arithmetic) which is guaranteed to contain the point at which the derivative should be evaluated. Iterative procedures can be specified which limit the growth of the width of the resulting interval.

Since this method depends on the ability of the computer to evaluate higher-order derivatives of f, it is handy to have a computer program which can do analytic differentiation. Such computer programs have indeed been written, either tailored for the purpose at hand [9], or in more general settings, such as the FORMAC capability for the FORTRAN IV compiler on the IBM 7094.

The success of interval arithmetic in this setting is somewhat difficult to evaluate. The problem is that for reasonably complex systems of equations and for long ranges of integration with respect to the independent variable, the resulting interval tends to be too wide to be of much practical value. Attempts at elaborate transformations to reduce the error growth due to the remainder term evaluation being too crude have in general been defeated by the fact that the structure of interval arithmetic (lack of additive inverses) causes growth of widths of intervals due to too many operations. Also, on some computers (such as the CDC 1604) the floating-

point hardware structure of the computer is so unfriendly that interval arithmetic operations are rather time-consuming. For short integrations, and for qualitative estimates, interval arithmetic may be very valuable.

ROOTS OF POLYNOMIALS. Moore suggested that a simple procedure for localizing zeroes of rational functions can be developed using interval arithmetic. Such a procedure was indeed programmed [3]. The method is based on the simple fact that if  $P$  is given rational form in  $n$  variables,  $R$  a rectangle in  $E^n$ , and  $P(R)$  evaluated in interval arithmetic does not contain the point 0, then  $P$  (as a function of real variables) cannot possibly have any zeroes in  $R$ .

An iterative procedure can be implemented based on the fact that if  $R_1$  and  $R_2$  are two rectangles in  $E^n$  each of which contains a given zero of  $P$ , then their intersection must necessarily also contain that zero. Thus, an extension of Newton's method is possible, as long as care is taken at each iteration to intersect the new interval (which may not be contained in the one obtained at the previous iteration) with the old one, thus guarding against divergence. This is called by Moore "the method of interval contractions". Clearly any such procedure must converge, but the limit will in general be an interval, rather than a point. If the limit interval is too wide, the process may be repeated by subdividing the original rectangle  $R$  into smaller ones.

Similar results were obtained for the complex domain (Boche [2] having extended the concept of interval arithmetic to the complex plane) by Hansen [6] and Bennett [1].

For this problem, interval arithmetic may well be the best (computationally speaking) method of obtaining results, especially if it is desirable to find regions guaranteed not to contain any zeroes of some given function.

MATRIX INVERSION AND THE EIGENVALUE PROBLEM. The problem of inverting matrices in the context of interval arithmetic comes from two distinct sources. Problem one: given a matrix with real elements, obtain a (real) inverse with automatic error bounding of round-off. Problem two: given a method of obtaining solutions of some problem in ordinary arithmetic (for example, Newton's method in  $n$  variables) which calls for inverting matrices, extend this method to the case where interval arithmetic will be used for the solution (possibly because the coefficients are only approximately known). That is, in problem two we are asked to invert a matrix with interval elements.

Since it is not a priori clear what we mean by an "inverse" of an interval-valued matrix, we define this inverse to be the set of inverses of all of the real matrices contained in the given interval matrix. It is understood that the inverse is defined only if the interval matrix does not contain any singular real matrices.



Hansen ([4] and [5]) has worked extensively on this problem. He shows that a direct extension of the standard methods for matrix inversion (such as modifications of Gauss - Seidel) to interval arithmetic is not very useful, because of the many arithmetic operations involved, and (again) because of the lack of additive inverses. Instead, he develops several methods, all based on essentially the same principle. What he does is to compute an (approximate) real inverse of the real center of the interval matrix, and then (using some iterative procedure) compute in interval arithmetic bounds for the width of each element of the true inverse of the interval matrix. The variations in the iterative procedures consist of trying to represent things in such a way as to have as many terms as possible be non-interval.

Similar considerations apply to the problem of finding eigenvalues and associated eigenvectors of real-valued or interval-valued matrices. Again, direct extensions of the standard techniques used for real arithmetic are not satisfactory. Hansen [6] suggests iterative procedures using interval arithmetic once approximate solutions are obtained using real arithmetic.

The numerical results quoted by Hansen suggest that very good accuracy can be obtained using interval arithmetic. His methods do converge, although he does not discuss the rate of convergence. Note that in Hansen's methods it frequently pays to carry out the real computations involved using extended-precision arithmetic, since in general multiple-precision arithmetic is much faster than the interval arithmetic procedures required, and it is worthwhile to go to great lengths to save an iterative step.

SYSTEMS OF SIMULTANEOUS EQUATIONS. Let  $f(x)$  denote the set of  $n$  rational forms  $f_i(x)$  in the  $n$  formal variables  $x_j$ , and let it be desirable to find a solution to  $f(x) = 0$  in the vicinity of some point  $x_0$  in  $E^n$ . A method proposed by Moore goes as follows.

Let  $y$  be a solution near  $x_0$ ; i.e. let  $f(y) = 0$ . (Of course, we do not know  $y$  explicitly.) Expanding  $f(x)$  as a Taylor series with remainder about  $y$ , we have  $f(x_0) = f(y) + (x_0 - y)J(z)$ , where  $z$  is some point "between"  $x_0$  and  $y$ , and  $J$  is the Jacobian matrix evaluated at  $z$ . Expressing  $z$  as  $y + \theta(x_0 - y)$ , where  $\theta$  is a vector with elements between 0 and 1, it can be seen that if  $R$  is a rectangle which contains both  $x_0$  and  $y$ , then  $R$  also contains  $z$ . Hence, we can try to solve  $(x_0 - y)J(R) = f(x_0)$  for  $y$ . This will yield a new rectangle  $R'$  which contains  $y$ , and which can then be intersected with  $R$  to yield a (hopefully) smaller rectangle  $R''$ . We now solve  $(R'' - y)J(R'')$  for  $y$ , etc; this will eventually converge to (we hope) a small interval containing the real solution  $y$ .

Hansen gives a slight improvement in the method [6]; this is essentially a slightly better way of writing things down for computation.



It can be seen that this is a variant of Newton's method, adapted for interval arithmetic. It requires that  $f$  contain no other zeroes near the point in question, for otherwise the Jacobian  $J$  becomes singular. Again it pays to obtain as precise an initial guess as possible, using ordinary (possibly extended-precision) arithmetic.

The author knows of no numerical experimentation with solving large systems of equations using interval arithmetic.

THE TWO-POINT BOUNDARY-VALUE PROBLEM. This problem is currently under investigation by Hansen. He has devised a general method for tackling the solution of

$$y^{(n)} = f(x, y, \dots, y^{(n-1)})$$

with a total of  $n$  conditions prescribed at the end points  $x = 0$  and  $x = 1$ . His method, based on an adaptation of a finite-difference method, gives sharp bounds at the mesh points and less sharp bounds throughout the interval. It will be described in a forthcoming paper.

OTHER POSSIBLE APPLICATIONS. Interval arithmetic may have potentially many uses. It has been suggested that it can be used in control theory, where it is desirable to let parameters in differential equations range over certain restricted domains. Another potential area of utility is in design, where it can be used in conjunction with an on-line computer system. A designer, seated in front of a terminal in communication with a computer, can experiment with various possible designs by letting some variables range over a set of interval values. With instant feedback from the computer, the designer can begin to get a feel for the effects of perturbations in the design parameters. Using interval arithmetic in this setting is particularly attractive because sharp bounds are not required - the qualitative estimates would be produced in relatively little time, and would at the same time be completely rigorous, covering all possible cases.

THE REPRESENTATION PROBLEM. The major trouble with interval arithmetic is that due to the lack of inverses forms normally considered algebraically equivalent are computationally quite different. It is always advisable when using interval arithmetic to eliminate entirely expressions of the form  $x - x$ . Other reductions of this type suggest themselves.

The general problem can be stated as follows. Suppose that  $f$  is a given function (from  $E^n$  into the reals) and suppose that it is desired to obtain bounds on the range of values of  $f$  over some rectangle  $R$  using interval arithmetic. What is the "best" way of representing  $f$  from the point of view of obtaining the narrowest bound?

There are three different approaches to this problem. One can try to obtain an optimal representation for  $f$ . (The author strongly suspects that this approach is not in general workable; that is, given a general function  $f$ , there is no algorithmic procedure that would allow the selection of a "best" form.) A second approach can be based on the following: if

$f_1$  and  $f_2$  are two different representations for  $f$ , and  $f_1(R) = I_1$ ,  $f_2(R) = I_2$ , then  $I_1 \wedge I_2$  also contains the range of  $f$  over  $R$ . It may be possible by judiciously choosing among different representations for  $f$  to obtain successively better approximations to the range of  $f$ . Although there is probably no algorithmic procedure guaranteed to converge for an arbitrary function  $f$ , it may be possible to find some programmable heuristics which greatly reduce growth of interval widths. The third approach consists of subdividing the original rectangle  $R$  into smaller rectangles and performing the required evaluations on each of the small pieces. This process will generally result in narrower bounds, and is in fact guaranteed to converge to the exact range of  $f$  regardless of the representation chosen. The convergence is however so slow compared to the overhead for repeating the computations for each one of the smaller intervals that this approach is not very practical.

Moore has noticed that a certain representation, which he calls the centered form, will frequently yield good results. Briefly, this scheme goes as follows: Given a formal function  $f$  of (say) one variable  $x$ , and assuming that we are interested in evaluating  $f$  over the interval  $[a,b] = [m - \frac{1}{2}(b-a), m + \frac{1}{2}(b-a)]$ , we represent  $f$  as expanded about the midpoint  $m$ . That is, we obtain a form  $g$  by the relation  $g(x-m) = f(x) - f(m)$ , so that  $g(y) = f(y+m) - f(m)$ .  $g$  has to be represented in the most "economical" way possible, so that the number of occurrences of the term  $y$  cannot further be reduced. Since  $f([a,b]) = g([- \frac{1}{2}(b-a), \frac{1}{2}(b-a)])$  we have moved the required interval evaluation to be centered about zero.

For an example, let  $f(x) = x - x^2$ , and let the interval in question be  $[0,1]$ . The actual range of values of  $f$  is of course  $[0, \frac{1}{4}]$ . Evaluation of  $f$  as written yields  $[0,1] - [0,1] * [0,1] = [0,1] - [0,1] = [-1,1]$ . Writing  $f$  in "nested" form as  $x*(1-x)$  yields  $[0,1] * [0,1] = [0,1]$ ; an improvement, but still not very good. Writing  $f$  in centered form, we have (with  $y = x - \frac{1}{2}$ )  $g(y) = -y^2 + \frac{1}{4}$ , so that  $f(x)$  is represented as  $-(x - \frac{1}{2})^2 + \frac{1}{4}$ ; interval evaluation of this form yields  $-[-\frac{1}{2}, \frac{1}{2}] * [-\frac{1}{2}, \frac{1}{2}] + \frac{1}{4} = [0, \frac{1}{4}]$ . This turns out the best that can be done for any given representation with the evaluation of only one interval. If however we are willing to evaluate separately the range of  $f$  on  $[0, \frac{1}{2}]$  and also on  $[\frac{1}{2}, 1]$ , then by using the centered form it turns out that we can bound the range of  $f$  by  $[0, 3/8]$ . In fact, if we keep halving the width of the (equal) intervals, it can be shown that interval evaluations approach the upper bound  $\frac{1}{4}$  linearly with the width.

Lest the reader conclude that the centered form is always the best representation, consider the function  $f(x) = x + x^2$ , and let the interval in question be  $[2-s, 2+s]$  where  $0 \leq s \leq 2$ . Then both straightforward interval evaluation and the nested form give  $[s^2-5s+6, s^2+5s+6]$ , which is the exact range of values. In centered form, however, we represent  $f$  as  $(x-2)(x+3)+6$ ; evaluation of this yields  $[-s^2-5s, s^2+5s]+6$ , which exceeds the actual width by  $2s^2$ .

It is possible (and desirable) to modify the rules of interval arithmetic in order to reduce spurious growth of intervals. One obvious and

easily programmable change is to define, for all intervals  $I$ ,

$$I^n = \{x^n : x \in I\}$$

This in general yields smaller intervals than the computation of  $I_1 * I_2 * \dots * I_n$  for  $I_1 = I_2 = \dots = I_n = I$ . Other modifications of this sort, which take into account known and easily computable exact ranges of values of a set of elementary common forms, may improve the performance (and possibly even speed up the operation of the system, as generally fewer multiplications will have to be performed during the computations).

Note that with changes of this sort, some of the properties of interval operations no longer hold. For example, with the change indicated above for raising to powers, subdistributivity no longer holds in its original form; the interval  $I*(I+1)$  need no longer be contained in the interval  $I^2 + I$  (whether it is or not depends on  $I$ ). If  $I = [-1,1]$ , then

$$I*(I+1) = [-1,1] * [0,2] = [-2,2];$$

while

$$I^2 + I + [0,1] + [-1,1] = [-1,2].$$

This tends to complicate the representation problem even further, since it becomes desirable to have a representation contain as many (in some sense) as possible of the forms whose ranges of values are exactly computable. The changes are all for the better, however; the complications result because we now have better ways of representing functions than formerly.

SYSTEMS PROGRAMMING FOR INTERVAL ARITHMETIC. Programming for interval arithmetic is somewhat similar to writing (general real) computational routines in the early days of computing, before the hardware implementation of floating-point arithmetic. At level 1, the systems programmer has to build the basic tools for performing interval computations: an adder, a multiplier, an inverter for producing an interval  $(1/d, 1/c)$  given the interval  $(c,d)$ , and (if exponentiation is desired) functions that compute good bounds on the range of values of the EXP and LOG operators. (Similarly, other elementary transcendental functions such as SIN should be incorporated.)

At level 2, tools must be provided for convenient interfacing with the user. For a simple example: subtraction can obviously be implemented very simply using the adder of level 1; at the same time, it is clearly not desirable to have the user perform this implementation every time he wishes to execute subtraction. Thus, a set of subroutines must be provided for the user which he can conveniently call. There are likely to be a large number of such subroutines, for the following reason. It is generally desirable to allow the user to mix the mode of the variables freely; he should be allowed to add a integer-valued variable or constant to an

integer-valued one. By the time all possible combinations of modes for operands are accounted for, the number of different subroutines provided is staggering. (Actually, there are typically about eight different routines, each of which has many entry points.)

It is clear that any such package of subroutines should be FORTRAN compatible. While the level 1 subroutines usually have to be written in machine language, there is usually no reason why the level 2 routines themselves cannot be written in the FORTRAN language.

The representation of interval numbers within a computer for FORTRAN might have been quite awkward were it not for the fact that formally an interval number looks just like a complex number. Any FORTRAN language compiler equipped to handle complex numbers can be tricked into handling interval numbers by the appropriate TYPE declarations. This is very handy for getting interval numbers in a decent format into and out of the computer, and also for defining interval-valued constants. (Arrays of interval numbers are also easier to handle if they are defined as being of TYPE COMPLEX.)

The arithmetic operations have to be performed by calls to the appropriate routines. Some computers (for example, the CDC 1604 and 3600) have a feature in their FORTRAN compilers which allow the definition of other (non-standard) variable types. What this means is that the compiler, when it encounters a variable of non-standard type, generates a call automatically to the appropriate arithmetic routine. This simplifies usage of interval arithmetic greatly, since the user, once he defines a variable as being of TYPE INTERVAL, can use it in statements as if it were any other type (integer or real). In fact, should this prove desirable, it is possible to define variables as being of type "double-precision interval" (the appropriate routines would have to be provided). For an example of an interval-arithmetic package of the sort just described, see [8].

The level 2 routines will depend to some extent on the exact working of the FORTRAN compiler. The level 1 routines are essentially compiler-independent; they are however heavily dependent on the way the given computer performs floating-point operations. (For convenience of interfacing with FORTRAN, the interval endpoints should usually be represented as floating-point numbers.) The (real) operations have to be performed at each end point in roughly the sequence: 1) perform the operation in a double length accumulator by using both the A and the Q registers without rounding; 2) normalize the result; 3) round to a single-precision floating-point number by adding (or subtracting) a 1 in the last place, unless the result was exact. If the computer does not allow this sequence of operations to be performed using the hardware floating-point instructions, then these operations have to be simulated by software, using fixed-point instructions.

Similar considerations apply to the computation of the transcendental functions. The functions should be computed in such a way that the result is off by at most one in the least significant bit of the single-precision answer.

Exponentiation can be implemented using the LOG and EXP routines. The system should however first determine if the exponent is an integer (even if represented as a floating-point number). As indicated, a substantial reduction in the growth of the widths of intervals can be effected if integer exponentiation is computed by repeated multiplications, using the true-range-of-values for raising to powers.

REFERENCES. The first place any interested reader should look is Moore [7]; aside from its definitive nature, it contains a rather complete bibliography of relevant literature. For a more up-to-date list, see Bennett [1].

- [1] G.K. Bennett, Jr. "A Method for Locating the Zeros of a Polynomial using Interval Arithmetic." Report published by the Computer Center, Texas Technological College, June 1967.
- [2] R.E. Boche, "Complex Interval Arithmetic with some Applications" Unpublished Master's Thesis, San Jose State College, 1966.
- [3] R.H. Dargel, F.R. Loscalzo, and T.H. Witt. "Automatic Error Bounds on Real Zeros of Rational Functions." Communications of the ACM, Vol. 9, Number 11, Nov. 1966.
- [4] E.R. Hansen, "Interval Arithmetic in Matrix Computations, Part I." SIAM Journal on Numerical Analysis, Series B, Vol. 2, Number 2 (1965).
- [5] E.R. Hansen and R. Smith, "Interval Arithmetic in Matrix Computations, Part II." SIAM Journal on Numerical Analysis, Series B (to appear).
- [6] E.R. Hansen, "On Solving Systems of Equations Using Interval Arithmetic." Mathematics of Computation (to appear).
- [7] R.E. Moore, Interval Analysis. Prentice-Hall, Inc. 1966.
- [8] A. Reiter, "Interval Arithmetic Package: INTERVAL". MRC Library Program #2, Mathematics Research Center, University of Wisconsin.
- [9] A. Reiter, "Automatic Generation of Taylor Coefficients: TAYLOR". MRC Library Program #3, Mathematics Research Center, University of Wisconsin.

## HOMEOSTATIC ORGANIZATIONS FOR ADAPTIVE PARALLEL PROCESSING SYSTEMS

Robert M. Dunn  
U. S. Army Electronics Command  
Fort Monmouth, New Jersey

An effective Army is not possible without the effective performance of tactical communications and information processing functions. An intriguing possible realization for the future is one which considers an integrated system providing service for both the communications and information processing functions. Within the realm of such a possibility, one may visualize a utility-like availability to these services for any qualified and authorized user.

At least three distinct approaches to such a military system are apparent. The first approach would provide each tactical element with a separate facility for the integrated services. The second approach would be to have many tactical elements time share a central facility. And the third approach would be to provide an Army-wide, common-user network for the integrated services. This network would be designed to tolerate losses of parts of itself without serious degradation of service from the remaining balance.

From a technological point of view, the separate facility approach is clearly the most near term and expedient. However, in the long term, this approach suffers from two weaknesses. First, either each facility is tailored to each tactical element or a single type of overly general, excessively capable facility is designed for all needs. Neither alternative is very desirable. The second weakness is that the set of separate facilities must be embedded in a superfacility to provide the basis for interchange of information between functionally distinct, but organizationally unified tactical elements.

The centralized, time-sharing approach implies minimal equipment costs and simplified logistics. This approach also provides ample opportunity for the just cited information interchange. However, this centralized approach guarantees chaos, not to mention severe losses and possible defeat, in the event of the destruction of such a facility. The mere hint of its existence would assure that such a facility became a prime target.

The merits and demerits of the network approach are not as readily compared and balanced against each other. Technologically, the network approach is the least certain. Economically, it is possibly as expensive or more expensive than the most costly already considered. Technological



turns will establish the degree of the logistics problems it presents, and so on. But, all of the real or anticipated uncertainties or drawbacks are potentially balanced or surpassed by the potential advantages of this approach. It could increase operational flexibility. It could enhance tactical survivability. The quality of service would be greatly improved. Such an approach could even foster a design that permits dynamic system growth and/or adaptation to changing requirements and/or applications and/or environments.

However, a great deal of knowledge is not available on network processor systems. This scarcity is the cause of our uncertainty about the network approach to the integrated system. Therefore, the objective of this discussion is to enlarge our generalized understanding of a network which is primarily composed of digital processors, information storage sub-systems, and other special or limited purpose sub-systems. For example, analog processor, hybrid processors, communications equipments, weapons systems, etc. This integrated tactical utility is considered to be geographically dispersed and offers the following features:

- Each subscriber approaches the system, uniformly, as a common-user, whether it be for communications or information processing services.
- Automatic control of the system is operationally distributed across the nodes of the network.
- The system automatically determines which aspects of itself are necessary to satisfy each user's service request by analyzing each service request. The system then automatically allocates and interconnects the necessary resources if, and from wherever, they are available within the network.
- Multiple users may simultaneously access the system without incurring mutual interference to the limit of the systems' capacity.
- Lastly, arbitrary subsets of the users of the system may cooperate via the system, using it as their means of inter-connection and basis for cooperation.

The most important implication of these features is the set of items that must be considered as separately allocatable resources within the system. Such usual things as computer programs, storage capacity, information, communications, sensors, and processors are within this set. But, atypically, this set includes the control function or even other users!



Our formulation of the system relies on two assumptions. The first is that for any system of the type under consideration, there exists a positive integer  $N$  such that the system is said to be an  $N$ th level organization. This implies that there are  $N$  hierarchical levels of structure where lower level functional elements are combined to form higher level functional elements. These combinations may either be permanent or temporary for some transient functional purpose.

The second assumption is that every output of every functional element is an input to some other functional element. Therefore, the inputs and the outputs are members of the same set of symbols.

In addition to these assumptions, there are a number of constraints upon the formulation of the system.

First, the behavior of the system must be able to be characterized such that all functional elements of a given type have identical physical realizations. The interests of economics and logistics are the motivation.

Second, the control function must imply neither a centralized organization, nor an omniscient attitude towards the system's status nor a large amount of status information or transmission thereof. Otherwise, the survivability objective would be immediately obviated. Next, the control function must allow for a non-deterministic allocation of resources. When resources are probabilistically allocated as the result of a search, the degree of omniscience and the amount of status information necessary to the control function may be drastically diminished.

Another constraint upon the formulation is that the notion of a control function must be limited to explicit control only of a node over itself. Each node of the system must neither require direction from nor be required to give direction to other nodes in the system. Implicitly, nodes may affect the behavior of each other by generating undirected service requests. Enhanced survival is the principal motivation for this constraint.

The final restriction is of a slightly different type. System effectiveness requires that there is a careful delineation of the operational and information environments along with the actual functional sequences to be performed. System efficiency requires that these delineators not be overly specific. The implication here is that such systems as we are discussing ought not to be programmed in the usual sense. That is, the development of a step-by-step sequence of directions is not the role of the user. The user, instead, specifies two things. On the one hand he declares the name or sequence of names of the function or functions to be performed. And on the other, he denotes the environmental and data references

germaine to these functions. The user then accepts whatever implementation is open to the system which will both satisfy these specifications and adhere to whatever priorities or time requirements that may be in effect. This "functional programming" approach is feasible because of the limited category of functional classes to which the military user is usually constrained. Therefore, although the digital processors within the system may be capable of emulating a Universal Turing machine, their actual pragmatic use will be limited to a well-defined set of interpretations. For example, they may be microprogrammed in some very gross sense.

Towards stating the model, consider, now, that an arbitrary abstract entity known as an organization has two major components: the structure and the behavior. Also consider that control is another abstraction interwoven into the fabric of the organization. The purpose of control is to assure that the behavior is achieved within the confines of the structure according to conditions imposed by the environment in which the organization exists. Finally, consider that control, structure, and behavior are further related in that the range of possible choices for any one of them is severely constrained by the previously chosen ranges for the other two. In fact, even after determining these three sets of possibilities, it will usually be the case that just a few of the possible combinations will be reasonable to consider according to various criteria.

If the term "system" is now considered to be the operational equivalent of "organization" then the set of primitive characteristics identifies the range of possible structures as that which also includes conventional telecommunications networks. More precisely, the set of structures are those partially describable as three-dimensional, coordinate arrays. These arrays are characterized by two properties. First, elements of the network need not exist at every coordinate intersection of the array. Second, interconnections only exist between elements of the network according to some appropriate functional, temporal, topological or metric definition of "nearness", i. e., those which are close together in some well-defined sense.

In turn, the set of possible behaviors is that which also includes the performance of arbitrary communications and information processing functions on a time-shared basis. A more precise statement would be that the set of behaviors are those partially describable as arbitrary sequences of any of transmission/reception, modulation/demodulation, multiplexing/demultiplexing, switching, data manipulation or computation functions. For any sequence or element of a sequence two properties hold. First, the system may not be continuously active in the response to that sequence or one of its elements. And second, for any functional module

of the system and any two consecutive, even contiguous, periods of activity of that module, the functional module needs not be active in response to the same sequence, or element thereof, in its successively active periods.

Finally, the set of possible controls is that which also includes the ability of local sections of the network to be self-managing. The definition of "local" is dynamically determined, in time, according to the magnitude of the response required by arbitrary service request. Again towards precision, the set of controls are those partially describable as mappings from the cross product set of the set of stimulators or inputs with the set of functional elements onto the set of sub-structures of the organization. Each of these sets of structure, behaviors, and controls is very comprehensive.

Via these notions of "organization," "structure," "behavior," and "control," there exists a precise context in which to formulate the model which hopefully will exhibit an ability to select some optimal combination of members of the three sets. In so doing, the model must allow for a functionally modular system which degrades gracefully and which can dynamically alter its own active internal organization. The model must, for the sake of generality, also allow for a homogeneous system as regards process, structure, and behavior. By this we mean that the abstract characterizations of either gross purpose, gross structure, or gross behavior of any functional element at any Kth level of the system is isomorphic to the corresponding abstractions for arbitrary functional elements at the same or different levels of the system.

We now make the following definitions:

- Definition 1. Functional Element - an instance of a separately allocatable system.
- Definition 2. Change Requirement - an input to some functional element of the system.
- Definition 3. Configuration - a set of interconnected functional elements and a description of that interconnection.
- Definition 4. Transformation Rule - an operator on the set of configurations.
- Definition 5. Response - a change requirement generated by the activity of a functional element.

We now let

$S_n$  = an  $n^{\text{th}}$  level organization;

$a_i$  = the  $i^{\text{th}}$  type change requirement;

$M_k^{\Delta_j^k}$  = A functional element at level K of the system of type  $\Delta_j^k$ .  $j$  ranges over the set of element types which are possible at level K;

$\Sigma_h$  = a type of configuration where the set of configurations is given by the set of graphs whose members are models for interconnection schema within the system;

$\Gamma_{a_i}$  = a transformation rule induced by an  $i^{\text{th}}$  type change requirement;

$F$  = the control function;

$H$  = the response function.

Any system may then be defined as an ordered sextuple.

$$S_n = \{A, C, G, D, F, H\}$$

where  $A = \{a_1, a_2, \dots, a_p\}$ . The set of types of change requirements;

$C = \{\Sigma_1, \Sigma_2, \dots, \Sigma_q\}$  the set of types of configurations;

$G = \{\Gamma_{a_1}, \Gamma_{a_2}, \dots, \Gamma_{a_p}\}$  the set of types of transformation rules corresponding to the set of types of change requirements;

$D = \{\Delta^1, \Delta^2, \dots, \Delta^n, \dots, \Delta^w\}$  the set of sets of possible element types at each level of the system;  $\Delta^w$  implies that the system may become a more complex system up to W levels deep.

$F: \{A \times D\} \xrightarrow{\text{into}} \{G \times C\}$  the control function;

$H: (\Gamma_{a_i}, M_r^{\Delta_j^r}) \longrightarrow (M_{r+\beta}^{\Delta_k^{r+\beta}}, a_o)$  the response function where  $-1 \leq \beta \leq 1$  and is always an integer or zero.

It is seen that the control function is a mapping from the set of order pairs of change requirement types and element types into the set of order pairs of transformation rules and configuration types. It is also seen that the response function is the result of the application of a change requirement induced transformation rule to a functional element; more strictly speaking to a configuration of that functional element. The result is some new functional element and a generated change requirement. Here, we can interpret  $\beta = 0$  to mean that the new functional element is either the same as the old one or, at most, a reconfiguration of the components of the old functional element. Similarly, we may interpret  $\beta = 1$  to mean that the old functional element has been combined into a more complex element. Finally, we may interpret  $\beta = -1$  to mean that the new functional element is the result of some decomposition of the old functional element. In addition, the following is always true. If  $M$  is a functional element at the  $K^{\text{th}}$  level and is of type  $\Delta_j^k$  then we may say that

$$M_k^{\Delta_j^k} = \sum_{i=0}^{k-1} h_i M_i^{\Delta_i^i}$$

This states that, in general, each functional element is an interconnection of lower level functional elements arranged in one of the possible configurations. In particular, this is true of the system as a whole.

$$S_n = \sum_{i=0}^{n-1} M_i^{\Delta_i^i}$$

Finally, we note that the model is independent of concern for actual levels of system organization or echelons of users. We also note that the nature of the functional elements or their manners of implementation did not enter into the model. Therefore, in systems such as this we expect to be able to dynamically juggle arrangements, relationships, or interconnections between functional elements as diverse as trunk group frames in a time division multiplex system, the multiplexors themselves, operating procedures, or even entire nodes. At the same time, the forms of realization of these functional elements may be as varied as an information stream, a message, wired logic, stored logic, or even stored program.

In practical terms, a network such as depicted in Figure 1 may range over many hundreds of square miles. The movement of users and equipment within this area appears to be best served by the class of systems discussed herein. Such networks would necessarily employ random search techniques for locating individual users within its domain.

As a consequence, the typical node in such a network may itself be a network of the same class as depicted in Figure 2. Again random search techniques would be utilized for routing traffic within the node. In turn, possible depictions of typical processor and memory modules appear in Figure 3 and 4 respectively. Further details on such practical considerations may be found in the brief bibliography at the end of the paper.

In summary, we have been talking about a network of processors which controls its own active interconnection scheme, dynamically regulates the distribution of load across itself in order to achieve an equilibrium state, and does all of this without a central scheduler or controller!

Borrowing from the physiologist, we shall label the drive towards an equilibrium state, the "homeostatic" aspect of our system and claim that its realization is a function of the organization which characterizes the system. The alteration of temporal and functional relationships between nodes in the network in response to new functions or service requests we take as an ability to alter behavior and so label the system "adaptive." The parallelism in the system is readily apparent. Therefore, in general, we have been talking about homeostatic organizations for adaptive parallel processing systems.

#### BIBLIOGRAPHY

1. Hambrock, H. Svala, G. G. and Prince, L. J., "Saturation Signalling - Toward Optimum Alternate Routing", 9th National Communication Symposium, Utica, N. Y., October 1963.
2. Cave, W. C. and Dunn, R. M., "Saturation Processing: An Optimized Approach to a Modular Computing System", Tech. Report 2636, USAECOM, Fort Monmouth, New Jersey, November 1965.
3. Dunn, R. M., "Modular Organization for Nodes in an Adaptive Homeostatic Process - Oriented System", Tech. Report 2722, USAECOM, Fort Monmouth, N. J., June 1966.
4. Dunn, R. M., "Extended Time-Sharing Systems", Tech. Report 2806, USAECOM, Fort Monmouth, N. J., January 1967.

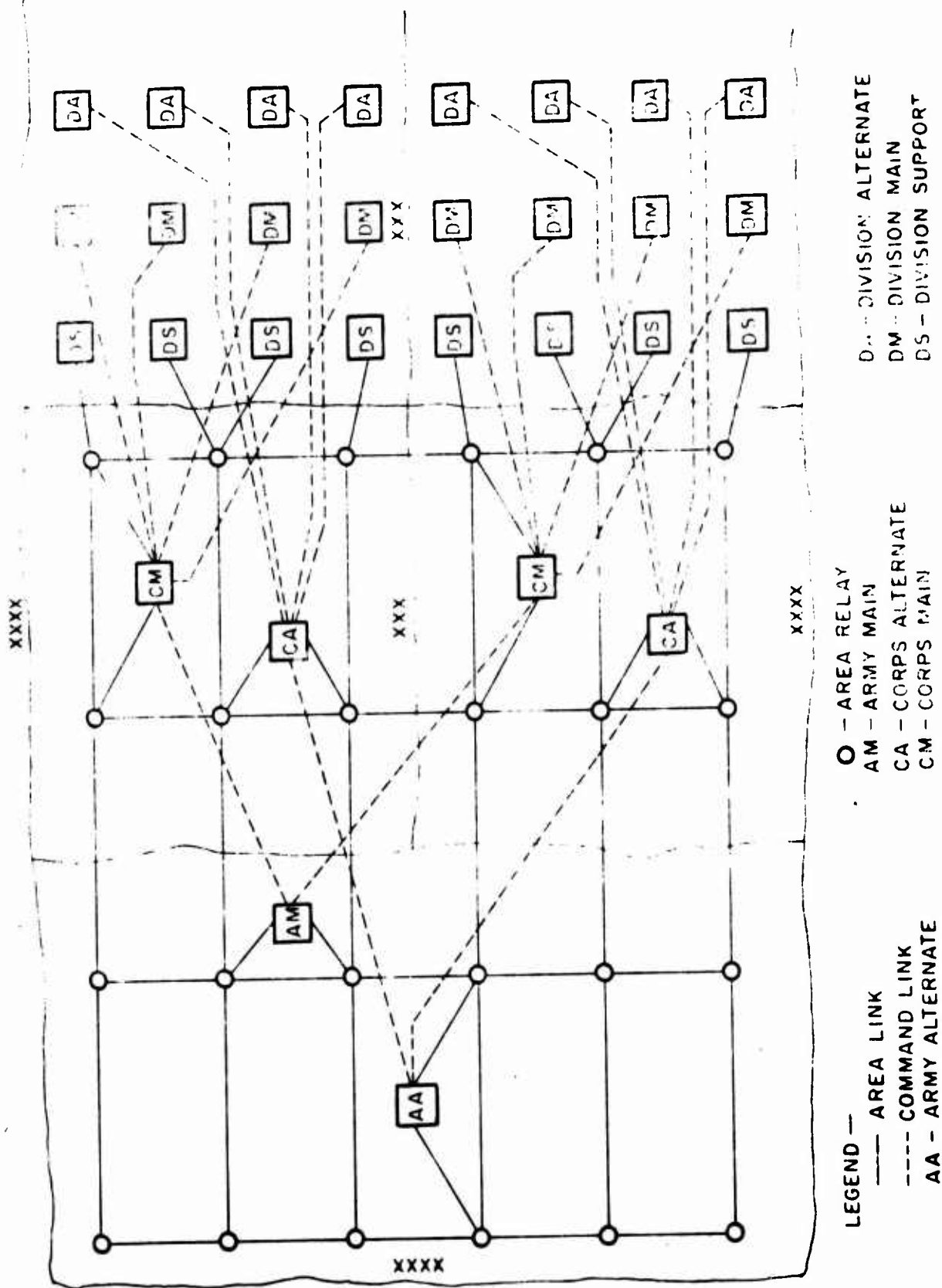


FIGURE I THEORETICAL FIELD ARMY NETWORK



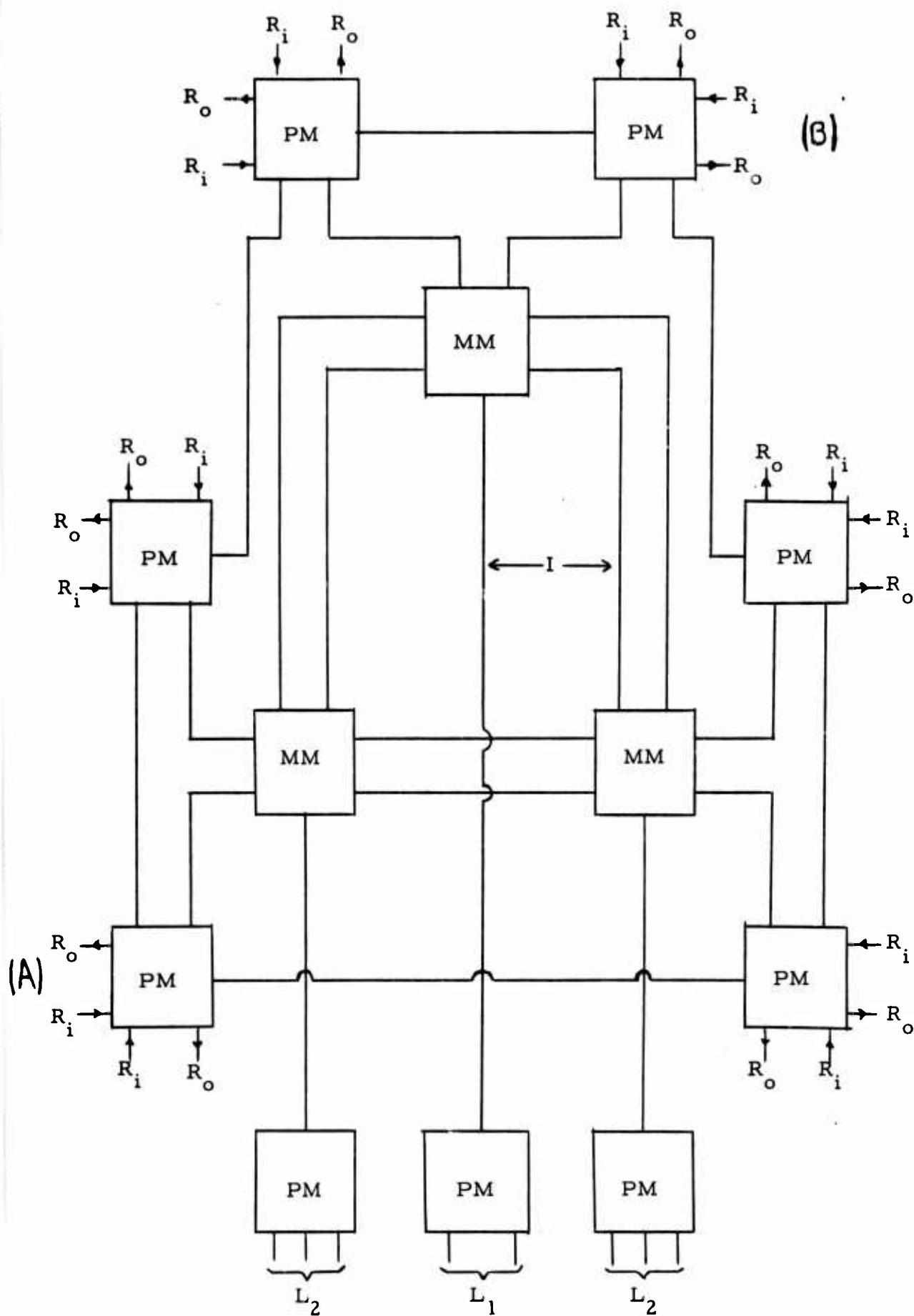


FIGURE 2. Typical Node for about 300 Channels



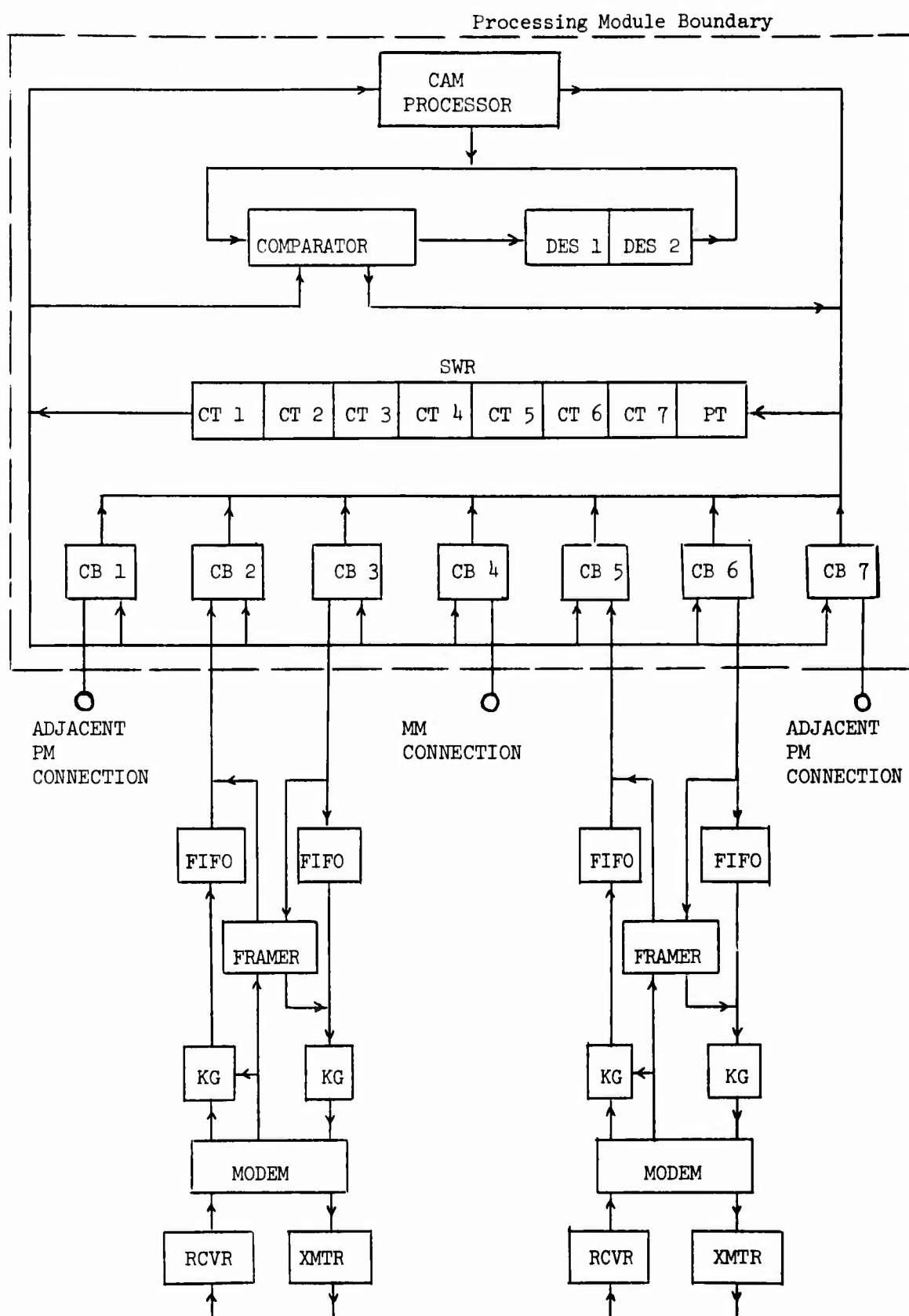


FIGURE 3: Typical Processor Module & Related Modules

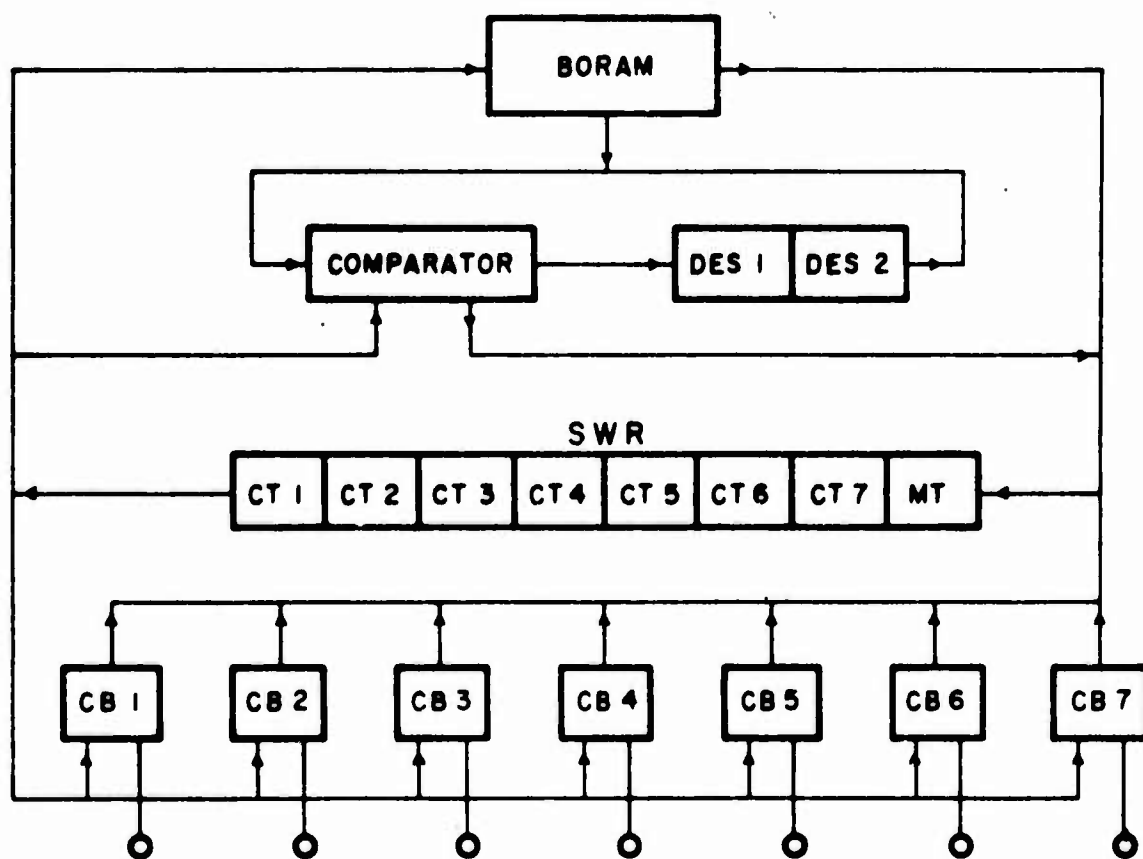


Figure 4 TYPICAL MEMORY MODULE

## LOGICAL STRUCTURE OF AN AUTOMATICALLY SEQUENCED EXPLOSIVE CONTROL DEVICE

Sylvan H. Eisman  
Pitman-Dunn Research Laboratories  
U. S. Army Frankford Arsenal

**ABSTRACT.** The question to be considered here concerns the interconnection of a set of one-shot devices which are to be activated in one of several predetermined sequences. Selection of the first device is made externally. In addition to performing its own action, each device initiates a pulse which travels along an explosive cord or MDC line to activate the next device in the sequence or to break another linking explosive cord. The essence of the problem is to define a procedure for interconnecting all required sequences so that one and only one will operate correctly when properly initiated. This is done by setting up a connection matrix representing all the sequences and then, by various operations on it, determining which links are to be broken and by what devices. This gives a solution but not an optimal one. Suggestions are made for improving individual solutions. An example is carried through the entire discussion and a computer program which mechanizes the procedure is exhibited as an appendix.

**INTRODUCTION.** The question to be considered here concerns the interconnection of a set of one-shot devices which are to be activated in one of several predetermined sequences. Selection of the first device is made externally. In addition to performing its own action, each device initiates a pulse which travels along an explosive cord or MDC line to activate the next device in the sequence or to break another linking explosive cord. The essence of the problem is to define a procedure for interconnecting all required sequences so that one and only one will operate correctly when properly initiated.\*

**STATEMENT AND DISCUSSION OF PROBLEM.** Given a set of  $n$  devices,  $d_1, d_2, \dots, d_n$ , it is required to interconnect them by explosive cords so that various preselected sequences of these devices will be actuated. Explosive cords for all sequences must be present at the initial installation of the devices and the final choice of sequence is made at the time of operation by selecting the starting point for the required chain of events.

\*Properties of MDC lines, methods for construction of the devices; possible application and other questions concerned with the physical realization of the system will be discussed elsewhere [1].

Example: Given devices a, b, c, d and e, it is required to be able to actuate them in any one of the four sequences abcde, cbade, bdac or d, upon external command. The first sequence will require MDC lines to carry a pulse from a to b, \* from b to c, etc; the third will need MDC lines from b to d, from d to a, and a to c. For the last sequence, only d is to operate and no other MDC lines are needed (or permitted).

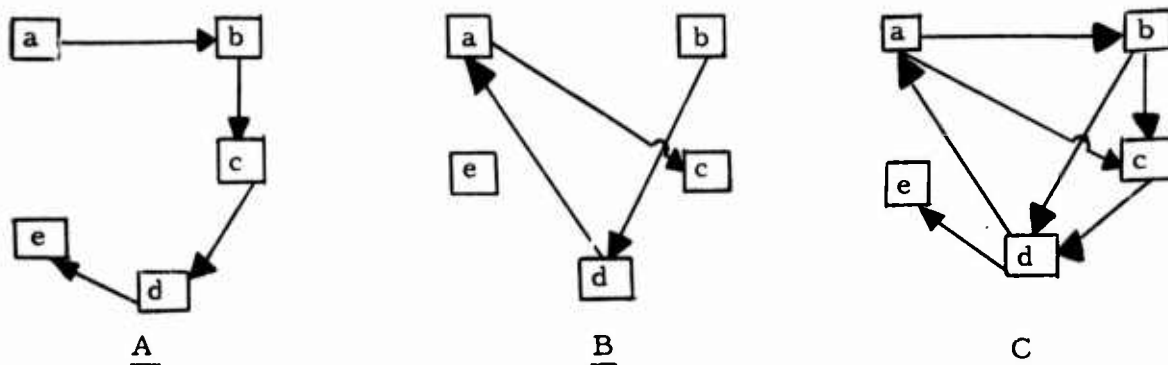


Figure 1

Figures 1A and 1B diagram the first and third sequences of the example. Figure 1C combines the connections for the two cases.

However, combining all the sequences does away with the definition of a unique successor to each device and special precautions must be taken to eliminate unwanted paths in the explosive chain. This can be done by destroying (negating) certain pathways by means of other exploding cords.

Example: In Figure 1C, if device a is chosen as the starting point, implying\*\* sequence abcde, the explosive pulse will travel to c as well as to b and interfere with proper operation of the system. To assure correct sequencing, MDC lines (a, c) and (b, d) would have to be cut. MDC line (d, a) can remain intact since element a has already functioned by the time element d is activated.

\*In this discussion the MDC lines or explosive cords will be treated as being unidirectional. The bidirectional case will be considered in a later section.

\*\*It should be noted that each sequence must start with a different element. For if two started with the same element, abcd and acd, for example, some external action must take place to indicate which of the two has been selected. This external event is then actually part of the system and should be labeled as the first device.

Example: Logically, it is relatively simple to cut (b, d) in the first case by having a initiate the cutting action (by means of another exploding cord, for example). It is not so simple to remove (a, c) because the pulse to c may get thru before the pulse to destroy the connection line does since both emanate from a. To obviate this, we make the

Assumption: It is possible to construct the equipment so that MDC destroyers (negators) act before all other MDC lines emanating from the same device.

Even though several devices might be available for breaking a MDC line, the strategy here will be to cut it as early as possible in the sequence. That is, activation of the first device will cut away all MDC lines which will interfere with the operation of its particular sequence.

It may happen that a negator which is essential to correct operation of one sequence interferes with proper operation of another. The negator must itself be broken by another explosive cord (second negation) during operation of the other sequence.

Example: Figure 2A shows the diagram of 1C with the addition of the two negators to permit sequence abcde to function properly. For sequence bdac, line (b, d) will function properly (Line (b, c) is to be ignored for the purpose of this particular explanation) since the negator from a \* has not yet been activated. However, line (a, c) will be broken by the negator from a before it performs its function since, by the assumption, the pulse travels faster along a negator than along a connecting line. That negator must be

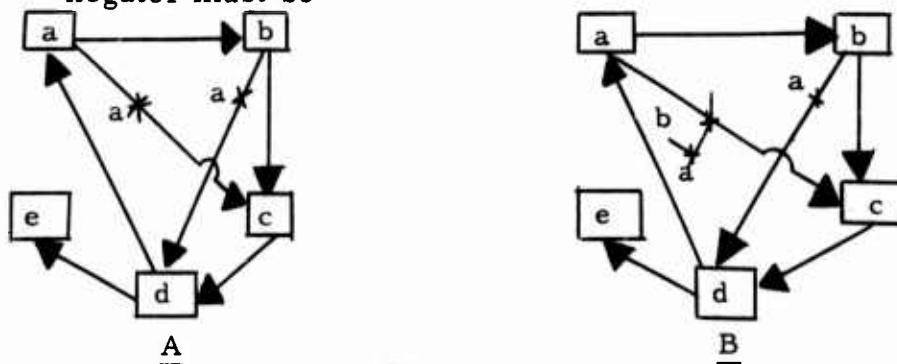


Figure 2

\*The negators from a to break (a, c) and (b, d) are shown as crosses on the lines. The point of origin of a negator (and, later, second negator) is shown close to the cutting point to avoid excessive lines in the diagram.

broken before it can function in sequence bdac. This can be done, as shown in Figure 2B, by a second negation from either b or d and the first element is chosen to simplify the procedure.

Will negations of an order higher than the second be required? That is, will there be occasions where it is necessary to break a second negation? The answer is no, as the following informal argument shows. Doing away with a second negation implies that the negator which it breaks has become necessary in some third sequence or that its original purpose has been interfered with. The latter case is impossible since negators are chosen to act from the first element in a sequence and the other problem can be bypassed by having a separate negator for each different sequence requiring it. If a different strategy were chosen for the origin of negators it is quite possible that a higher order negator would be needed.

SOLUTION. Let  $n$  be the number of devices which have been denoted as  $d_q$ ,  $1 \leq q \leq n$ , and which are to be arranged into  $m \leq n$  operating sequences. To simplify the notation we shall drop the symbol  $d$  and use the index  $q$  as the label. Thus each operative sequence is represented as a sequence of integers. We now form an  $m \times n$  sequence matrix,  $S$ , as follows: each of the  $m$  sequences of integers will form a row of  $S$ , where the order of the rows is arbitrary; if any row has less than  $n$  integers, sufficient 0's are added on the right to bring the number up to  $n$ .

Example: given six devices labelled 1, 2, ..., 6, with the following required sequences: 316524, 4312, 54321, 654321, 123456, the following 5 X 6 sequence matrix can be formed:

$$S = \begin{pmatrix} 3 & 1 & 6 & 5 & 2 & 4 \\ 4 & 3 & 1 & 2 & 0 & 0 \\ 5 & 4 & 3 & 2 & 1 & 0 \\ 6 & 5 & 4 & 3 & 2 & 1 \\ 1 & 2 & 3 & 4 & 5 & 6 \end{pmatrix}$$

Associated with each  $S$  matrix is an  $n \times n$  connection matrix  $C$  whose entries  $c_{r,t}$  are 1 if for some sequence there is an MDC line from device  $r$  to device  $t$  and 0 otherwise. These lines will also be referred to as major connectors to differentiate them from first and second negators.

Example: associated with S above is the 6 x 6 matrix:

$$C = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Each sequence will be identified by its first element. For any sequence k of length p, the rows and columns of C can be permuted until the first p rows and columns are in the same order as in the sequence. Label this permuted matrix  $C_k$ . The last n-p rows will not be needed but the last n-p columns will, although their order is irrelevant. Note that the row labels represent the required devices and that the column labels represent all the existing devices

Example:

	3	1	6	5	2	4
3		1			1	1
1			1		1	
6				1		
5			1		1	1
2	1	1				1
4	1			1		

$C_3$

	5	4	3	2	1	6
5		1		1		1
4	1		1			
3		1		1	1	
2		1	1		1	
1			1			1

$C_5$

The following observations can be made on the  $C_k$ . The first p-1 elements on the first superdiagonal represent the major connectors required for the proper functioning of sequence k. There are no 1's on the main diagonal since no device is connected to itself.

For each required device, the corresponding row contains a 1 in a column where ever there is a connection from it to another device. The 1's below the main diagonal represent connections to device already activated and will be of no interest here. Those above the first superdiagonal represent connections to devices which can still be activated but which must be prevented from operating at this time.

Finally, if  $p < n$ , there must be no connection remaining from device  $p$  to any other still active device. This leads to the formulation of

**Rule 1:** A sufficient set of negations for each sequence  $k$  is determined by all the 1's above the first superdiagonal in  $C_k$  as well as the  $p^{\text{th}}$  1 on that diagonal if it exists. In each case, the source of the negator is device  $k$ .

A more formal proof that this rule produces the desired negations will be found in Appendix I.

It will be convenient to record the negations in the  $C_k$  by encircling the 1's identified in Rule 1.

Example:

	3	1	6	5	2	4
3		1			(1)	(1)
1			1		(1)	
6				1		
5			1		1	(1)
2	1					1
4	1	1		1		

$C_3$

	4	3	1	2	5	6
4		1			(1)	
3	1		1	(1)		
1				1		(1)
2	1	1	1			

$C_4$

	5	4	3	2	1	6
5		1		(1)		(1)
4	1		1			
3		1		1	(1)	
2		1	1		1	
1				1		(1)

$C_5$

	6	5	4	3	2	1
6		1				
5	1		1		(1)	
4		1		1		
3			1		1	(1)
2			1	1		1
1	1				1	

$C_6$



	1	2	3	4	5	6
1		1				(1)
2	1		1	(1)		
3	1	1		1		
4			1		1	
5		1		1		1
6					1	

$C_1$

The five permutations of  $C$  of the example are shown and the negators for each sequence have been encircled in their respective matrices. In  $C_5$  line (1, 6) requires negations (even though on the first superdiagonal) to prevent 6 from being activated when it is not required in the sequence. Similarly, if an MDC line had existed from 2 to 5 (and/or 6),  $C_4$  would have shown the need for its negation. For each sequence  $k$ , the first device will be taken as the source for the negators.

Once the negations are determined for all the sequences, they can be combined and exhibited in the matrix  $C$  by encircling the 1's involved and labeling each circle with the index of the devices from which the negators must come. Call the matrix with this extra labeling  $C'$ .

Example:

	1	2	3	4	5	6 <sup>-1,4,5'</sup>
1		(1) <sup>-3</sup>				(1)
2	1		1	(1) <sup>-1</sup>		
3	(1) <sup>-5,6</sup>	(1) <sup>-3,4</sup>		(1) <sup>-3</sup>		
4			1		(1) <sup>-4</sup>	
5		(1) <sup>-5,6</sup>		(1) <sup>-3</sup>		(1) <sup>-5</sup>
6					1	

$C'$

Line (1, 6) is shown to need negations in sequences 1, 4, and 5 by the matrices  $C_1$ ,  $C_4$  and  $C_5$  respectively. Therefore, since the negation will come from the first device in each sequence the device names 1, 4 and 5 are appended as shown.

The information on negators can also be condensed into tabular form as is actually done, but in slightly different format, in the computer procedure.

Example:

MDC LINE	NEGATOR FROM DEVICE
(1, 2)	3
(1, 6)	1, 4, 5
(2, 4)	1
(3, 1)	5, 6
(3, 2)	3, 4
(3, 4)	3
(4, 5)	4
(5, 2)	5, 6
(5, 4)	3
(5, 6)	5

$C'$ , which now incorporated information on negators as well as major connectors, can be used to determine a set of second negators. This matrix can be permuted, as was  $C$  to form  $C'_k$  with the first  $p-1$  elements in the first superdiagonal indicating not only the major connectors required for sequence  $k$  but also those negators capable of interfering with its proper operation.

Example:

	3	1	6	5	2	4
3		① <sup>3,6</sup>			1	1
1			① <sup>1,4,5</sup>		1	
6				1		
5			1		① <sup>5,6</sup>	1
2	1	1				① <sup>1</sup>
4	1			1		

$$\underline{C'_3}$$

Since only negators involving required major connectors for sequence  $k$  are of interest here, the others are not shown.

Let  $N_{w,d}$  denote the negator of major connector  $w$  from device  $d$ . If  $d$  appears in sequence  $k$  after the origin of  $w$ , the negator will not interfere with the proper functioning of the system. If  $d$  is the origin of  $w$  or appears before it in sequence  $k$ , then the negator will cut the required major connector before it can operate. This leads to

Rule II: To find a sufficient set of second negations for each sequence  $k$ , consider  $C'_k$ . Provide second negations for all those negators  $N_{w,d}$  which affect the first  $p-1$  elements on the first superdiagonal and for which the row labeled  $d$  does not follow the row in which the negator in question appears. In each case, the origin of the second negator is device  $k$ .

A more formal proof that this rule actually provides the necessary control over the negations is also contained in Appendix I.

Example: Consider  $C'_3$ .  $w = (3,1)$  is negated by both devices 5 and 6. Since columns labeled 5 and 6 follow column 3, no second negation is required. On the other hand  $w = (5,2)$  required second negations for the negators from 5 and 6 since neither of these two columns follows column 2. The second negation, discovered in  $C'_3$ , comes from device 3. For  $w = (1,6)$  a second negation is required for the negator from 1 while none is required for those from 4 and 5.

The following list is an extension of the previous one to show second negations. They are in parentheses behind the negators they affect.

MDC LINES	NEGATORS AND SECOND NEGATORS
(1, 2)	3(4)
(1, 6)	1(3), 4, 5
(2, 4)	1(3)
(3, 1)	5, 6
(3, 2)	3(5, 6), 4(5, 6)
(3, 4)	3(1)
(4, 5)	4(1)
(5, 2)	5(3), 6(3)
(5, 4)	3
(5, 6)	5(1)

**SIMPLIFICATIONS.** It is possible for redundancies to exist among the negators and second negators. That is, since negating devices have been chosen as the first in the sequence, it is conceivable that another device after the first is already acting satisfactorily as a negator. In this case, the number of negators and/or second negators is reducible.

Example: Consider the negators from devices 5 and 6 for line (3, 1). In sequence 6, device 5 precedes device 3. Therefore, 5 can provide the negation and the one from 6 can be eliminated. The same idea justifies the removal of two second negations in (3, 2), the ones from device 6 to the negators from 3 and 4. The connections to the affected MDC lines now appear as:

(3, 1)	5
(3, 2)	3(5) 4(5)

Another possibility for reducing the number of negating lines is to delay the action until the last possible moment. That is, if line (a, b) is negated by a, c, d, ... i, device a might serve in all cases and c, d, ... i could be eliminated since, by the assumption on page 2, negating pulses always travel faster than pulses along regular MDC lines. The procedure will not always work if the negator from i has a second negation on it.

Example: Consider the negators from devices 1, 4 and 5 to line (1, 6) and ignore for the moment, the second negator from 3 to 1. Then device 1 is sufficient to negate (1, 6) in the sequences beginning with 4 and 5 and those two negators would be eliminated. However, as is actually the case, the second negation from 3 in both sequences 4 and 5 would reach the negating line from 1 before 1 itself is activated. This would prevent proper negation of (1, 6) in these two sequences.

Example: MDC line (3, 2) is now negated by devices 3 and 4. In this case, the negation from 4 can be eliminated since device 5, which causes a second negation of 3 does not occur (at all) in sequence 4 in time to prevent proper negation.

These two simplification rules can be incorporated in the procedure to reduce the number of MDC lines.

Example: The present example can be simplified to provide a smaller number of connections.

MDC LINES	NEGATORS AND SECOND NEGATORS
(1, 2)	3(4)
(1, 6)	1(3), 4, 5
(2, 4)	1(3)
(3, 1)	5
(3, 2)	3(5)
(3, 4)	3(1)
(4, 5)	4(1)
(5, 2)	5(3)
(5, 4)	3
(5, 6)	5(1)

BIDIRECTIONAL CASE. In the Bidirectional case, a pulse may travel in either direction along an MDC line. Therefore, it sequence  $d_1 d_2 \dots d_{n-1} d_n$  is constructed with bidirectional lines, connections also exist along the path  $d_n d_{n-1} \dots d_2 d_1$ . These connections must be shown in the connection matrix  $C$  and, in practice this can be accomplished quite simply by deriving  $C$  from  $S$  as before and forming a new  $C$  equal

to  $C \cup C^T$ . \* Since the procedures in Rules I and II involve only operations on  $C$  no further changes have to be made to solve the problem for this case.

Example: Using the same sequences as in the previous example, we have

$$C = \begin{matrix} 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{matrix} \quad C^T = \begin{matrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 \end{matrix}$$

$$C \cup C^T = \begin{matrix} 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 \end{matrix}$$

The two rules can now be applied to this new connection matrix to give a list of negators and second negators.

MDC LINES	NEGATORS AND SECOND NEGATORS
(1, 2)	3(4, 5, 6)
(1, 3)	1, 5, 6
(1, 6)	1(3), 4, 5, 6
(2, 3)	3(5, 6), 4(5, 6)
(2, 4)	1(3), 4(3), 5(3), 6(3)
(2, 5)	1(3), 4, 5(3), 6(3)
(3, 4)	3(1)
(4, 5)	3(1), 4(1)
(5, 6)	5(1)

\*  $C^T$  is the transpose of  $C$ .  $C \cup C^T$  means the new matrix has a 1 in position  $i, j$  if  $C$  and/or  $C^T$  have a 1 in position  $i, j$  and 0 elsewhere.

Several redundancies can be removed as in the unidirectional case. More care must be taken, however, since a line indicated for example, as (2, 4) now means a connection from 4 to 2 as well as one from 2 to 4.

Example: The above negators and second negators can be reduced to the following:

MDC LINES	NEGATORS AND SECOND NEGATORS
(1, 2)	3(4)
(1, 3)	1, 5
(1, 6)	3, (5)
(2, 3)	1(3), 4
(2, 5)	1(3), 4, 5(3)
(3, 4)	3(1)
(4, 5)	3(1), 4(1)
(5, 6)	5(1)

VALIDATION PROCEDURE. The above solution guarantees proper operation of the sequences barring, of course, blunders in the application of the rules. A FORTRAN program, supposedly doing away with this latter possibility, is used to generate the first and second negators and its listing appears in Appendix II.

The introduction of simplification procedures which have been neither formalized nor mechanized raises the possibility of introducing logical errors as well as blunders into the solution. It is therefore advisable to check that these changes still produce the required sequences. This can be done by considering the revised list of major connectors and first and second negators and following the sequence of actions after the required initial devices are activated. The procedure is straightforward: for each device, activate the second negators it controls, then the still active first negators and then the still active major connectors. If more than one major connector is left from the activated device, there is an error. If only one major connector is left, the next element in the sequence is identified, and the procedure repeated for it. If no connectors are left, the sequence is ended. A FORTRAN program (also appearing in Appendix II) has been written to mechanize this procedure and print out the valid sequences. Ambiguities which result in improper functioning are also indicated.

SUMMARY. A procedure has been demonstrated for generating a set of negators and second negators which is sufficient for proper functioning of the required sequences. It does not produce an optimal solution in the sense of minimizing the total number of connections although, in individual cases, redundancies can be eliminated. Whether or not an effective general procedure for minimizing the connections (short of enumerating all possible combinations and selecting the smallest) exists is unknown at this time.



## APPENDIX I

The following shows that Rule I produces a sufficient set of negators.

### PROOF:

We have to show that the negators given by the rule prevent operation of any device out of sequence.

Consider the required sequence  $k$  of length  $p$ :  $k_1, k_2, \dots, k_p$ .  $C_k$  has been derived from  $C$  by means of the permutation

$$P_k = \begin{pmatrix} k_1 & k_2 & \dots & k_p \\ 1 & 2 & & p \end{pmatrix} \cdot \begin{pmatrix} n-p+1 & \dots & n \\ n-p+1 & \dots & n \end{pmatrix} \quad 1 \leq p \leq n$$

where  $n$  is the number of devices. There is a connection from device  $k_i$  to  $k_j$  ( $1 \leq i, j \leq n$ ) if and only if  $c_k(i, j) = 1$  in  $C_k$ .

The rule calls for negation of all major connectors represented by those  $c_k(i, j)$  for which  $j > i+1$  ( $1 \leq i \leq p \leq n-2$ ) as well as those for which  $i = p$  and  $j = i+1$ . This is to be done by negators originating from device  $k = k_1$ . Since negators act before major connectors emanating from the same device the connectors described above are effectively non-existent for this sequence and we may replace their representations  $c_k(i, j)$  by 0, forming the new matrix  $C''_k$ .

Now consider any required device  $k_i$  ( $1 \leq i_0 \leq n$ ) and remember that it can initiate another device  $j$  if and only if  $c''_k(i_0, j) = 1$ .

If  $j < i_0$ ,  $k_j$  has functioned before  $k_{i_0}$  and the presence of a major connector is of no consequence.

The case  $j = i_0$  does not occur since no device is ever connected to itself.

Since for all  $j > i_0 + 1$   $c''_k(i_0, j)$  has been set to 0 (negated) by the rule, none of these  $k_j$  can be activated by  $k_{i_0}$ .

We are therefore concerned only with  $j = i_0 + 1$ , the first superdiagonal.

From the construction of  $C_k$  and  $C''_k$ ,  $c''_k(i_0, i_0 + 1) = 1$  for  $1 \leq i_0 \leq p-1$ .

If  $p=n$ , only the first  $n-1$  major connectors of the sequence remain and the sequence functions properly.

If  $p < n$ ,  $c''_k(i_o, i_o + 1) = 0$  for  $i_o = p$  since the rule calls for negation in this case. Therefore, there is no connection from  $k_p$  to any other device and the sequence terminates as it should.

The following shows that Rule II provides second negators which prevent unwanted first negators from interfering with the required major connectors. It also demonstrates that the required sequence functions properly.

PROOF:

For this rule, we must show:

A. that no extraneous device functions out of sequence since the second negators might conceivably destroy first negators given by Rule I.

B. that the second negators do, in fact, prevent first negators from interfering with the required major connectors.

1. For any required sequence  $k$ , consider  $C'_k$  which shows all the system negators found by repeated application of Rule I.
2. All first negations that are required in sequence  $k$  are initiated by device  $k$  (the first element in the sequence), and these are all off the first superdiagonal of  $C'_k$ .
3. By Rule II, second negators from device  $k$  affect only the first  $p-1$  elements which lie on the first superdiagonal of  $C'_k$ .
4. When sequence  $k$  is called for all first and second negators from the initial device,  $k$ , function before anything else. However, by (2) and (3) it can be seen that none of the first negators used in the sequence are destroyed by second negators from  $k$ . By a verbatim repetition of the proof used for Rule I, it is now seen that no extraneous device functions out of sequence.
5. The first  $p-1$  elements on the first superdiagonal of  $C'_k$  represent the major connectors which must function for sequence  $k$  to operate properly. Consider any one of these

elements, say  $c'_k(i_o, i_o+1)$ , which has negators on it.

Let  $d$  be the origin of one of its negators. If  $d$  follows  $k_{i_o}$  in the sequence, the major connector functions properly before the negator is initiated. If  $d$  precedes  $k_{i_o}$  in the sequence or is  $k_{i_o}$ , the second negator from device  $k$  (provided by Rule II) eliminates the first negator before it can destroy the required major connector.

6. Since the quantities  $d$ ,  $i_o$  and  $k$  in the above were all arbitrary, the argument holds for all the sequences.

## APPENDIX II

The Program for the mechanization of the solution was written in FORTRAN II and run on a UNIVAC SS - 90 card system. Several comments concerning the procedures and conventions are necessary.

1. The FORTRAN listing can serve as its own flow chart. It contains several main, non-overlapping parts (the prefix N is used to identify integer arrays):
  - a. Set up sequence matrix NS
  - b. Set up connection matrix NC
  - c. Determine first negators
  - d. Determine second negators
2. It was found convenient to include all existing devices (not only the required ones) in NS. The non required ones follow the sequence and each identifier is preceded by a minus sign. Thus, the sequence 4 3 1 2 of our example is actually entered as 4 3 1 2 -5 -6. The order of the added devices is unimportant. This turned out to be a convenient way to signal the routine that, even though the sequence had ended, negators might still be required.
3. Rather than rearrange matrix NC to conform with each permutation, subscripted subscripts were used. That is, to select elements of NC for testing, we have to look at individuals NC (I,J) in a specified order. These orders are given by the rows of NS which contain the required sequences. I and J are both functions of the elements in NS:  $I = NS(\alpha, \beta)$   $J = NS(\gamma, \delta)$  for arbitrary  $\alpha, \beta, \gamma, \delta$ . Therefore, given  $\alpha, \beta, \gamma, \delta$ , we can find NC (I,J) as
$$NC(NS(\alpha, \beta) NS(\gamma, \delta))$$
4. Information on each negator is stored in an array indexed by the symbol NG. For each NG, as this array is being formed, another 5-position sector is cleared for use in storing the origins of at most 5 second negators. Should the number of negators and associated second negators exceed the arbitrary numbers of 50 and 5, respectively, one "DO" statement will have to be changed in addition to the 'Dimension' statement.
5. Throughout the program, several array elements which are used more than once have been renamed without subscripts. This was done to speed up the processor at the cost of, what is hoped to be, small decrease in readability.

```

C   FORTRAN PROGRAM # 541
C   XPLOSIVE COMPUTER  SCHLACK/EISMAN          LB500
      DIMENSION NS(20,20),NC(20,20),NREG      (50),NEND(50),NI(50),NUM(50),NSEC(50,5)
      COMMON NS,NC,NREG,NEND,NI,NUM,NSEC
C   SET UP SEQUENCE MATRIX NS
C   ONE SEQUENCE PER CARD, EVERY 4 SPA      CES, REMEMBER LEADING ZERO FOR ONE DIGIT
C   NUMBERS STARTING 3, 7,11,15,19,
C           23,27,31,35,39,43,47,51,55,59,63,67,71,75,79,83,87,91,95,99,103,107,111,115,119,123,127,131,135,139,143,147,151,155,159,163,167,171,175,179,183,187,191,195,199,203,207,211,215,219,223,227,231,235,239,243,247,251,255,259,263,267,271,275,279,283,287,291,295,299,303,307,311,315,319,323,327,331,335,339,343,347,351,355,359,363,367,371,375,379,383,387,391,395,399,403,407,411,415,419,423,427,431,435,439,443,447,451,455,459,463,467,471,475,479,483,487,491,495,499,503,507,511,515,519,523,527,531,535,539,543,547,551,555,559,563,567,571,575,579,583,587,591,595,599,603,607,611,615,619,623,627,631,635,639,643,647,651,655,659,663,667,671,675,679,683,687,691,695,699,703,707,711,715,719,723,727,731,735,739,743,747,751,755,759,763,767,771,775,779,783,787,791,795,799,803,807,811,815,819,823,827,831,835,839,843,847,851,855,859,863,867,871,875,879,883,887,891,895,899,903,907,911,915,919,923,927,931,935,939,943,947,951,955,959,963,967,971,975,979,983,987,991,995,999,1003,1007,1011,1015,1019,1023,1027,1031,1035,1039,1043,1047,1051,1055,1059,1063,1067,1071,1075,1079,1083,1087,1091,1095,1099,1103,1107,1111,1115,1119,1123,1127,1131,1135,1139,1143,1147,1151,1155,1159,1163,1167,1171,1175,1179,1183,1187,1191,1195,1199,1203,1207,1211,1215,1219,1223,1227,1231,1235,1239,1243,1247,1251,1255,1259,1263,1267,1271,1275,1279,1283,1287,1291,1295,1299,1303,1307,1311,1315,1319,1323,1327,1331,1335,1339,1343,1347,1351,1355,1359,1363,1367,1371,1375,1379,1383,1387,1391,1395,1399,1403,1407,1411,1415,1419,1423,1427,1431,1435,1439,1443,1447,1451,1455,1459,1463,1467,1471,1475,1479,1483,1487,1491,1495,1499,1503,1507,1511,1515,1519,1523,1527,1531,1535,1539,1543,1547,1551,1555,1559,1563,1567,1571,1575,1579,1583,1587,1591,1595,1599,1603,1607,1611,1615,1619,1623,1627,1631,1635,1639,1643,1647,1651,1655,1659,1663,1667,1671,1675,1679,1683,1687,1691,1695,1699,1703,1707,1711,1715,1719,1723,1727,1731,1735,1739,1743,1747,1751,1755,1759,1763,1767,1771,1775,1779,1783,1787,1791,1795,1799,1803,1807,1811,1815,1819,1823,1827,1831,1835,1839,1843,1847,1851,1855,1859,1863,1867,1871,1875,1879,1883,1887,1891,1895,1899,1903,1907,1911,1915,1919,1923,1927,1931,1935,1939,1943,1947,1951,1955,1959,1963,1967,1971,1975,1979,1983,1987,1991,1995,1999,2003,2007,2011,2015,2019,2023,2027,2031,2035,2039,2043,2047,2051,2055,2059,2063,2067,2071,2075,2079,2083,2087,2091,2095,2099,2103,2107,2111,2115,2119,2123,2127,2131,2135,2139,2143,2147,2151,2155,2159,2163,2167,2171,2175,2179,2183,2187,2191,2195,2199,2203,2207,2211,2215,2219,2223,2227,2231,2235,2239,2243,2247,2251,2255,2259,2263,2267,2271,2275,2279,2283,2287,2291,2295,2299,2303,2307,2311,2315,2319,2323,2327,2331,2335,2339,2343,2347,2351,2355,2359,2363,2367,2371,2375,2379,2383,2387,2391,2395,2399,2403,2407,2411,2415,2419,2423,2427,2431,2435,2439,2443,2447,2451,2455,2459,2463,2467,2471,2475,2479,2483,2487,2491,2495,2499,2503,2507,2511,2515,2519,2523,2527,2531,2535,2539,2543,2547,2551,2555,2559,2563,2567,2571,2575,2579,2583,2587,2591,2595,2599,2603,2607,2611,2615,2619,2623,2627,2631,2635,2639,2643,2647,2651,2655,2659,2663,2667,2671,2675,2679,2683,2687,2691,2695,2699,2703,2707,2711,2715,2719,2723,2727,2731,2735,2739,2743,2747,2751,2755,2759,2763,2767,2771,2775,2779,2783,2787,2791,2795,2799,2803,2807,2811,2815,2819,2823,2827,2831,2835,2839,2843,2847,2851,2855,2859,2863,2867,2871,2875,2879,2883,2887,2891,2895,2899,2903,2907,2911,2915,2919,2923,2927,2931,2935,2939,2943,2947,2951,2955,2959,2963,2967,2971,2975,2979,2983,2987,2991,2995,2999,3003,3007,3011,3015,3019,3023,3027,3031,3035,3039,3043,3047,3051,3055,3059,3063,3067,3071,3075,3079,3083,3087,3091,3095,3099,3103,3107,3111,3115,3119,3123,3127,3131,3135,3139,3143,3147,3151,3155,3159,3163,3167,3171,3175,3179,3183,3187,3191,3195,3199,3203,3207,3211,3215,3219,3223,3227,3231,3235,3239,3243,3247,3251,3255,3259,3263,3267,3271,3275,3279,3283,3287,3291,3295,3299,3303,3307,3311,3315,3319,3323,3327,3331,3335,3339,3343,3347,3351,3355,3359,3363,3367,3371,3375,3379,3383,3387,3391,3395,3399,3403,3407,3411,3415,3419,3423,3427,3431,3435,3439,3443,3447,3451,3455,3459,3463,3467,3471,3475,3479,3483,3487,3491,3495,3499,3503,3507,3511,3515,3519,3523,3527,3531,3535,3539,3543,3547,3551,3555,3559,3563,3567,3571,3575,3579,3583,3587,3591,3595,3599,3603,3607,3611,3615,3619,3623,3627,3631,3635,3639,3643,3647,3651,3655,3659,3663,3667,3671,3675,3679,3683,3687,3691,3695,3699,3703,3707,3711,3715,3719,3723,3727,3731,3735,3739,3743,3747,3751,3755,3759,3763,3767,3771,3775,3779,3783,3787,3791,3795,3799,3803,3807,3811,3815,3819,3823,3827,3831,3835,3839,3843,3847,3851,3855,3859,3863,3867,3871,3875,3879,3883,3887,3891,3895,3899,3903,3907,3911,3915,3919,3923,3927,3931,3935,3939,3943,3947,3951,3955,3959,3963,3967,3971,3975,3979,3983,3987,3991,3995,3999,4003,4007,4011,4015,4019,4023,4027,4031,4035,4039,4043,4047,4051,4055,4059,4063,4067,4071,4075,4079,4083,4087,4091,4095,4099,4103,4107,4111,4115,4119,4123,4127,4131,4135,4139,4143,4147,4151,4155,4159,4163,4167,4171,4175,4179,4183,4187,4191,4195,4199,4203,4207,4211,4215,4219,4223,4227,4231,4235,4239,4243,4247,4251,4255,4259,4263,4267,4271,4275,4279,4283,4287,4291,4295,4299,4303,4307,4311,4315,4319,4323,4327,4331,4335,4339,4343,4347,4351,4355,4359,4363,4367,4371,4375,4379,4383,4387,4391,4395,4399,4403,4407,4411,4415,4419,4423,4427,4431,4435,4439,4443,4447,4451,4455,4459,4463,4467,4471,4475,4479,4483,4487,4491,4495,4499,4503,4507,4511,4515,4519,4523,4527,4531,4535,4539,4543,4547,4551,4555,4559,4563,4567,4571,4575,4579,4583,4587,4591,4595,4599,4603,4607,4611,4615,4619,4623,4627,4631,4635,4639,4643,4647,4651,4655,4659,4663,4667,4671,4675,4679,4683,4687,4691,4695,4699,4703,4707,4711,4715,4719,4723,4727,4731,4735,4739,4743,4747,4751,4755,4759,4763,4767,4771,4775,4779,4783,4787,4791,4795,4799,4803,4807,4811,4815,4819,4823,4827,4831,4835,4839,4843,4847,4851,4855,4859,4863,4867,4871,4875,4879,4883,4887,4891,4895,4899,4903,4907,4911,4915,4919,4923,4927,4931,4935,4939,4943,4947,4951,4955,4959,4963,4967,4971,4975,4979,4983,4987,4991,4995,4999,5003,5007,5011,5015,5019,5023,5027,5031,5035,5039,5043,5047,5051,5055,5059,5063,5067,5071,5075,5079,5083,5087,5091,5095,5099,5103,5107,5111,5115,5119,5123,5127,5131,5135,5139,5143,5147,5151,5155,5159,5163,5167,5171,5175,5179,5183,5187,5191,5195,5199,5203,5207,5211,5215,5219,5223,5227,5231,5235,5239,5243,5247,5251,5255,5259,5263,5267,5271,5275,5279,5283,5287,5291,5295,5299,5303,5307,5311,5315,5319,5323,5327,5331,5335,5339,5343,5347,5351,5355,5359,5363,5367,5371,5375,5379,5383,5387,5391,5395,5399,5403,5407,5411,5415,5419,5423,5427,5431,5435,5439,5443,5447,5451,5455,5459,5463,5467,5471,5475,5479,5483,5487,5491,5495,5499,5503,5507,5511,5515,5519,5523,5527,5531,5535,5539,5543,5547,5551,5555,5559,5563,5567,5571,5575,5579,5583,5587,5591,5595,5599,5603,5607,5611,5615,5619,5623,5627,5631,5635,5639,5643,5647,5651,5655,5659,5663,5667,5671,5675,5679,5683,5687,5691,5695,5699,5703,5707,5711,5715,5719,5723,5727,5731,5735,5739,5743,5747,5751,5755,5759,5763,5767,5771,5775,5779,5783,5787,5791,5795,5799,5803,5807,5811,5815,5819,5823,5827,5831,5835,5839,5843,5847,5851,5855,5859,5863,5867,5871,5875,5879,5883,5887,5891,5895,5899,5903,5907,5911,5915,5919,5923,5927,5931,5935,5939,5943,5947,5951,5955,5959,5963,5967,5971,5975,5979,5983,5987,5991,5995,5999,6003,6007,6011,6015,6019,6023,6027,6031,6035,6039,6043,6047,6051,6055,6059,6063,6067,6071,6075,6079,6083,6087,6091,6095,6099,6103,6107,6111,6115,6119,6123,6127,6131,6135,6139,6143,6147,6151,6155,6159,6163,6167,6171,6175,6179,6183,6187,6191,6195,6199,6203,6207,6211,6215,6219,6223,6227,6231,6235,6239,6243,6247,6251,6255,6259,6263,6267,6271,6275,6279,6283,6287,6291,6295,6299,6303,6307,6311,6315,6319,6323,6327,6331,6335,6339,6343,6347,6351,6355,6359,6363,6367,6371,6375,6379,6383,6387,6391,6395,6399,6403,6407,6411,6415,6419,6423,6427,6431,6435,6439,6443,6447,6451,6455,6459,6463,6467,6471,6475,6479,6483,6487,6491,6495,6499,6503,6507,6511,6515,6519,6523,6527,6531,6535,6539,6543,6547,6551,6555,6559,6563,6567,6571,6575,6579,6583,6587,6591,6595,6599,6603,6607,6611,6615,6619,6623,6627,6631,6635,6639,6643,6647,6651,6655,6659,6663,6667,6671,6675,6679,6683,6687,6691,6695,6699,6703,6707,6711,6715,6719,6723,6727,6731,6735,6739,6743,6747,6751,6755,6759,6763,6767,6771,6775,6779,6783,6787,6791,6795,6799,6803,6807,6811,6815,6819,6823,6827,6831,6835,6839,6843,6847,6851,6855,6859,6863,6867,6871,6875,6879,6883,6887,6891,6895,6899,6903,6907,6911,6915,6919,6923,6927,6931,6935,6939,6943,6947,6951,6955,6959,6963,6967,6971,6975,6979,6983,6987,6991,6995,6999,7003,7007,7011,7015,7019,7023,7027,7031,7035,7039,7043,7047,7051,7055,7059,7063,7067,7071,7075,7079,7083,7087,7091,7095,7099,7103,7107,7111,7115,7119,7123,7127,7131,7135,7139,7143,7147,7151,7155,7159,7163,7167,7171,7175,7179,7183,7187,7191,7195,7199,7203,7207,7211,7215,7219,7223,7227,7231,7235,7239,7243,7247,7251,7255,7259,7263,7267,7271,7275,7279,7283,7287,7291,7295,7299,7303,7307,7311,7315,7319,7323,7327,7331,7335,7339,7343,7347,7351,7355,7359,7363,7367,7371,7375,7379,7383,7387,7391,7395,7399,7403,7407,7411,7415,7419,7423,7427,7431,7435,7439,7443,7447,7451,7455,7459,7463,7467,7471,7475,7479,7483,7487,7491,7495,7499,7503,7507,7511,7515,7519,7523,7527,7531,7535,7539,7543,7547,7551,7555,7559,7563,7567,7571,7575,7579,7583,7587,7591,7595,7599,7603,7607,7611,7615,7619,7623,7627,7631,7635,7639,7643,7647,7651,7655,7659,7663,7667,7671,7675,7679,7683,7687,7691,7695,7699,7703,7707,7711,7715,7719,7723,7727,7731,7735,7739,7743,7747,7751,7755,7759,7763,7767,7771,7775,7779,7783,7787,7791,7795,7799,7803,7807,7811,7815,7819,7823,7827,7831,7835,7839,7843,7847,7851,7855,7859,7863,7867,7871,7875,7879,7883,7887,7891,7895,7899,7903,7907,7911,7915,7919,7923,7927,7931,7935,7939,7943,7947,7951,7955,7959,7963,7967,7971,7975,7979,7983,7987,7991,7995,7999,8003,8007,8011,8015,8019,8023,8027,8031,8035,8039,8043,8047,8051,8055,8059,8063,8067,8071,8075,8079,8083,8087,8091,8095,8099,8103,8107,8111,8115,8119,8123,8127,8131,8135,8139,8143,8147,8151,8155,8159,8163,8167,8171,8175,8179,8183,8187,8191,8195,8199,8203,8207,8211,8215,8219,8223,8227,8231,8235,8239,8243,8247,8251,8255,8259,8263,8267,8271,8275,8279,8283,8287,8291,8295,8299,8303,8307,8311,8315,8319,8323,8327,8331,8335,8339,8343,8347,8351,8355,8359,8363,8367,8371,8375,8379,8383,8387,8391,8395,8399,8403,8407,8411,8415,8419,8423,8427,8431,8435,8439,8443,8447,8451,8455,8459,8463,8467,8471,8475,8479,8483,8487,8491,8495,8499,8503,8507,8511,8515,8519,8523,8527,8531,8535,8539,8543,8547,8551,8555,8559,8563,8567,8571,8575,8579,8583,8587,8591,8595,8599,8603,8607,8611,8615,8619,8623,8627,8631,8635,8639,8643,8647,8651,8655,8659,8663,8667,8671,8675,8679,8683,8687,8691,8695,8699,8703,8707,8711,8715,8719,8723,8727,8731,8735,8739,8743,8747,8751,8755,8759,8763,8767,8771,8775,8779,8783,8787,8791,8795,8799,8803,8807,8811,8815,8819,8823,8827,8831,8835,8839,8843,8847,8851,8855,8859,8863,8867,8871,8875,8879,8883,8887,8891,8895,8899,8903,8907,8911,8915,8919,8923,8927,8931,8935,8939,8943,8947,8951,8955,8959,8963,8967,8971,8975,8979,8983,8987,8991,8995,8999,9003,9007,9011,9015,9019,9023,9027,9031,9035,9039,9043,9047,9051,9055,9059,9063,9067,9071,9075,9079,9083,9087,9091,9095,9099,9103,9107,9111,9115,9119,9123,9127,9131,9135,9139,9143,9147,9151,9155,9159,9163,9167,9171,9175,9179,9183,9187,9191,9195,9199,9203,9207,9211,9215,9219,9223,9227,9231,9235,9239,9243,9247,9251,9255,9259,9263,9267,9271,9275,9279,9283,9287,9291,9295,9299,9303,9307,9311,9315,9319,9323,9327,9331,9335,9339,9343,9347,9351,9355,9359,9363,9367,9371,9375,9379,9383,9387,9391,9395,9399,9403,9407,9411,9415,9419,9423,9427,9431,9435,9439,9443,9447,9451,9455,9459,9463,9467,9471,9475,9479,9483,9487,9491,9495,9499,9503,9507,9511,9515,9519,9523,9527,9531,9535,9539,9543,9547,9551,9555,9559,9563,9567,9571,9575,9579,9583,9587,9591,9595,9599,9603,9607,9611,9615,9619,9623,9627,9631,9635,9639,9643,9647,9651,9655,9659,9663,9667,9671,9675,9679,9683,9687,9691,9695,9699,9703,9707,9711,9715,9719,9723,9727,9731,9735,9739,9743,9747,9751,9755,9759,9763,9767,9771,9775,9779,9783,9787,9791,9795,9799,9803,9807,9811,9815,9819,9823,9827,9831,9835,9839,9843,9847,9851,9855,9859,9863,9867,9871,9875,9879,9883,9887,9891,9895,9899,9903,9907,9911,9915,9919,9923,9927,9931,9935,9939,9943,9947,9951,9955,9959,9963,9967,9971,9975,9979,9983,9987,999
```

```

DO 2: I#1:M
DO 3: J#1:N1
J1#J+1
IF (NS(I,J1)) 2:2:6
6 NC(NS(I;J),NS(I,J1))#1
IF (1-IND) 37:3:995
37 NC(NS(I,J1),NS(I,J))#1
3 CONTINUE
2 CONTINUE
C DETERMINE FIRST NEGATIONS
NG#1
DO 30: I#1:M
ISAVE#NS(I:1)
C DEPENDING ON COLUMN N+1 BEING 0
DO 31: J#1:N1
JANJ
NJ#NS(I;J)
IF (NJ) 30:996:10
10 IF (NS(I,J+1)) 9:997:8
9 JANJ-1
8 DO 32: K#JA+2:N
NNS#NS(I:K)
IF (NNS) 4:31:5
4 NNS#-NNS
5 IF (NC(NJ,NNS)) 998:32:7
7 NBEG(NG)#NJ
NEND(NG)#NNS
NI(NG)#ISAVE
NUM(NG)#1
DO 99: IJK#1:5
NSEC(NG,IJK)#0
99 CONTINUE
NG#NG+1

```

```

FOR EACH SEQUENCE
FOR EACH DEVICE
IF A DEVICE FOLLOWS
INSERT 1 INTO NC
UNI OR BI DIRECTIONAL
ALSO IN XPOSE IF BIDIRECTIONAL

```

```

INITIALIZE NEGATOR COUNTER
FOR EACH SEQUENCE
FIRST ELEMENT OF SEQ I
FOR EACH LINE
CHECK IF ELEMENT NEG OR POS.
IF OTHER END NEG., EXTRA NEGATION
SET BEGIN'G OF SEARCH BACK 1
LOOK FOR NEGATIONS ABOVE SUPERDIAGONAL
REVERSE SIGN NON-RQRD ELEM.
IF 1. REQUIRES NEGATION
BEGINNING NEGATED LINE
END NEGATED LINE
NEGATED BY DEVICE NUMBER
CLEAR SECOND NEGATOR VECTOR FOR
THIS NEGATOR

```

32 CONTINUE	
31 CONTINUE	
30 CONTINUE	
C DETERMINE SECOND NEGATIONS	
LL#NG-1	LIMIT NON NEGATORS SCANNED
DO 15, I#1,M	FOR EACH SEQUENCE
DO 15, J#1,N1	FOR EACH LINE
NBP#NS(I,J)	
NEP#NS(I,J+1)	
DO 13, L#1,LL	FOR EACH NEGATOR
IF (NBP-NBEG(L)) 96,12,96	CHECK BEGINNING
96 IF (1-IND) 97,13,995	UNI- OR BI- DIRECTIONAL
12 IF (NEP-NEND(L)) 13,14,13	CHECK END
97 IF (NBP-NEND(L)) 13,98,13	CHECK CONNECTOR IN
98 IF (NEP-NBEG(L)) 13,14,13	OTHER DIRECTION
14 DO 21, K#1,J	FOR ALL ELEMENTS BEFORE
IF (NI(L)-NS(I,K)) 21,22,21	DOES NEGATING INDEX APPEAR
22 NSEC(L,NUM(L))#NS(I,1)	SAVE SECOND NEGATOR
NUM(L)#NUM(L)+1	
GO TO 13	
21 CONTINUE	
13 CONTINUE	
15 CONTINUE	
DO 500, I#1,LL	
PRINT 300, I ,NBEG(I),NEND(I), NI(I),(NSEC(I,J),J#1,5)	
500 CONTINUE	
300 FORMAT (5X,15,5X,215,5X,15,5X,515)	
305 FORMAT (16X,9HFROM TO,8X,2HBY,5X ,14HSECOND NEG'S./)	
304 FORMAT (32X,3HNEG)	
303 FORMAT (2/)	
STOP	
998 PAUSE 998	
997 PAUSE 997	

996 PAUSE 996

995 PAUSE 995

END



Input (sequences)

3	1	6	5	2	4
4	3	1	2	-5	-6
5	4	3	2	1	-6
6	5	4	3	2	1
1	2	3	4	5	6

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

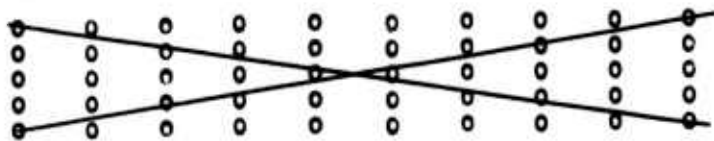
Output

	FROM	TO	NEG BY	SECOND	NEG'S.			
1	3	2	3	5	6	0	0	0
2	3	4	3	1	0	0	0	0
3	1	2	3	4	0	0	0	0
4	5	4	3	0	0	0	0	0
5	4	5	4	1	0	0	0	0
6	3	2	4	5	0	0	0	0
7	1	6	4	0	0	0	0	0
8	5	2	5	3	0	0	0	0
9	5	6	5	1	0	0	0	0
10	3	1	5	0	0	0	0	0
11	1	6	5	0	0	0	0	0
12	5	2	6	3	0	0	0	0
13	3	1	6	0	0	0	0	0
14	1	6	1	3	0	0	0	0
15	2	4	1	3	0	0	0	0

Unidirectional Example (IND = 1)

Input

3	1	6	5	2	4
4	3	1	2	-5	-6
5	4	3	2	1	-6
6	5	4	3	2	1
1	2	3	4	5	6



output

	FROM	TO	NEG BY	SECOND	NEG'S.			
1	3	2	3	5	6	0	0	0
2	3	4	3	1	0	0	0	0
3	1	2	3	4	5	6	0	0
4	5	4	3	1	0	0	0	0
5	4	2	4	0	0	0	0	0
6	4	5	4	1	0	0	0	0
7	3	2	4	5	6	0	0	0
8	1	6	4	0	0	0	0	0
9	2	5	4	0	0	0	0	0
10	5	2	5	3	0	0	0	0
11	5	6	5	1	0	0	0	0
12	4	2	5	3	0	0	0	0
13	3	1	5	0	0	0	0	0
14	1	6	5	0	0	0	0	0
15	6	1	6	0	0	0	0	0
16	5	2	6	3	0	0	0	0
17	4	2	6	3	0	0	0	0
18	3	1	6	0	0	0	0	0
19	1	3	1	0	0	0	0	0
20	1	6	1	3	0	0	0	0
21	2	4	1	3	0	0	0	0
22	2	5	1	3	0	0	0	0

Bidirectional / Example (IND = 2)

```

C    USS FORTRAN II *** VERSION 9000    22 JAN 63
C    COMPILED    7/28/67
C    FORTRAN PROGRAM # 509
C    4-28-67    SHE    L8500
C    XPLOSIVE COMPUTER- UNIDIRECTIONAL - VALIDATION PROCEDURE
C    ACCEPTS MAJOR CONNECTORS, NEGATORS AND SECOND NEGATORS IN
C    FORMAT OF OUTPUT FROM SOLUTION PROGRAM ( 541 )
C    ALL MAJOR CONNECTORS PRESENT, WHETHER NEGATED OR NOT,
C    MUST BE INCLUDED
C    PROGRAM ACCEPTS DEVICE NO. AND PRODUCES EITHER
C    1, THE UNIQUE SEQUENCE    OR
C    2, INDICATION OF AMBIGUITY
C
C    DIMENSION NBEG(50),NEND(50),NI(50),NSEC(50,5),NS(20)
C    COMMON NBEG,NEND,NI,NSEC,NS
C
C    B1#5HTOO L
C
C    B2#5HONG
C
C    C1#5HAMBIG
C
C    C2#5HUOUS
C
C    READ 900,NGM
C
C    DO 1,I#1,NGM
C
C    READ 901,NBEG(I),NEND(I),NI(I),(NSEC(I,J),J#1,5)
C
1    CONTINUE
C
101 READ 900,K
C
C    A1#5H
C
C    A2#5H

```

```

NS(I)NK
DO 2,IN2,20
NS(I)NO
2  CONTINUE
DO 3,IN1,NGM
NBEG(I)NABS(NBEG(I))
NI(I)NABS(NI(I))
3  CONTINUE
DO 20,KK2,20
NANO
DO 14,IN1,NGM
DO 13,JM1,5
IF (NSEC(I,J)) 910,14,11
11 IF (NSEC(I,J)-K) 13,12,13
12 IF (NI(I)) 14,910,112
112 NI(I)N-NI(I)
GO TO 14
13 CONTINUE
14 CONTINUE
DO 16,IN1,NGM
IF (NI(I)-K) 16,15,16
15 NBEG(I)N-NBEG(I)
16 CONTINUE
DO 19,IN1,NGM
IF (NBEG(I)-K) 19,17,19

```

```

17  NZ#NEND(I)
    DO 117,L#1,KK-1
    IF (NZ-NS(L)) 117,19,117
117  CONTINUE
    DO 317,II#1,NGM
    IF (NBEG(II)+K) 317,217,317
217  IF (NEND(II)-NZ) 317,19,317
317  CONTINUE
18  IF (NA) 910,218,118
118  IF (NA-NZ) 24,19,24
218  NA#NZ
19  CONTINUE
    IF (NA) 910,30,21
21  NS(KK)#NA
    K#NA
20  CONTINUE
    A1#B1
    A2#B2
    GO TO 30
24  A1#C1
    A2#C2
30  PRINT 902,A1,A2,(NS(I),I#1,KK)
    GO TO 101
900  FORMAT (I4)
901  FORMAT (5X,B15)

```

902 FORMAT (10X,2A5,20I4)

910 PAUSE 910

STOP

END

YYYOYYYYYY  
YYYOYYYYYY  
YYYOYYYYYY  
YYYOYYYYYY  
YYYOYYYYYY

211224195Y  
4004200054  
7100010400  
112225472Y  
122226372Y

(HEADERS)  
(HEADERS)  
(HEADERS)  
(HEADERS)  
(HEADERS)

#### REFERENCES

1. Schlack, A. F., Eisman, S. H., and Kowalick, J. F., "Explosive Control Devices" - Frankford Arsenal Report, in preparation

Additional information on these devices may be obtained in:

1. "Investigation of Propellant Actuated Devices for Use in Emergency Crew Escape Systems for Advanced Aerospace Vehicles," Phase III - Design Study, by T. H. Bleikamp, E. R. Lake, and D. R. McGovern of McDonnell Aircraft Corporation, Technical Report AFFDL-TR-65-26, Part II, April 1965, pages 49-53  
AD 646 738
2. "Mild Detonating Cord" Journal of the JANAF Fuze Committee, Serial #44.0, 3 May 1967.

## PROBLEM SOLVING BY DIGITAL-ANALOG SIMULATION\*

Howard M. Bloom  
Computation and Analysis Branch  
Harry Diamond Laboratories  
Washington, D. C.

**ABSTRACT.** An evaluation of four simulation languages, MIDAS, APACHE, MIMIC, and DSL/90, is made to determine their relative merits. The application of analog computer techniques to digital-analog simulation is considered. The problems discussed are as follows: solution to a set of linear algebraic equations, linear programming, hybrid simulation, partial differential equations, boundary value problems, parameter optimization using a least-squares error criteria, and roots of polynomial equations. A mathematical outline of the technique or problem is given as well as the digital program, written in DSL/90, which is used to represent the problem. Possible improvements in the simulation language are shown. Some of the suggestions presented include the ability to dimension variables, and a means of using an iteration technique.

---

\*This report will be published in full 1 January 1968 as TR-1357 of the Harry Diamond Laboratories.

PRECEDING  
PAGE BLANK



A SHELL COMPUTER PROGRAM WHICH DETERMINES THE  
PHYSICAL PROPERTIES OF AN ARTILLERY SHELL AND  
REPRESENTS ITS DIMENSIONS GRAPHICALLY

Forrest McMains  
Picatinny Arsenal, Dover, New Jersey

The purpose of this presentation is to describe a digital computer program which determines the physical properties of artillery shells and related items.

I have chosen to speak on this program for two main reasons:

First, the program is used daily at Picatinny Arsenal both in experimental design work and in the analysis of end items. Since it is used primarily by people who are not computer oriented extreme care had to be taken in writing the input-output operations. The input data had to be clear and concise. The output information had not only to be complete, including as many helpful and meaningful results as possible, but it also had to be kept brief.

Secondly, the reason for choosing this program concerns the manner in which it is able to handle large amounts of data in an almost error free manner. Special care has been taken so that every dimension of the shell (the input data to the program) can be enumerated logically and quickly. The resulting graph (which is nothing more than a picture of these dimensions) serves as an excellent check on the input values. A mere glance at the picture of the shell is usually sufficient to detect any input error. Further, and in most cases a final check for errors, consists in comparing this picture to the original blueprint of the shell.

An artillery shell is formally defined as a hollow projectile, designed to be given an explosive, a chemical or other filler and fired from a weapon. It is composed of body pieces (which are frustums of right circular cones and cylinders); ogive pieces (the curved, forward part of the projectile, including its pointed end) and fins (a fixed or adjustable airfoil attached to the projectile and parallel to the plane of symmetry which affords directional stability).

Each card of input to the program consists of the four to six dimensions of each piece plus an identification of this piece.

Figure 1 defines a body piece. Each body piece has four dimensions:

AB, the radius of the end closest to the reference axis;  
BB, the radius of the opposite end;

PRECEDING  
PAGE BLANK

HB, the length; and  
RB, the reference.

A reference axis must be chosen before any data is collected. Once this axis is selected, every shell piece must be referenced to it.

Two other parameters appear on the body item card of input: the density of the material used, and the identification of the item.

Figure 2 defines a fin item. Its dimensions are analogous to those comprising the body item: AF and BF are radii; RF is the reference; and HF is the length. Besides density and an identification, a third parameter, its thickness is also needed.

Figure 3 defines an ogival item. The parameters AV and BV are the X and Y coordinates of the origin of the ogival system of the system of the (circular) arc. RD is the radius of the arc; RV and HV are the reference and length values.

The three examples shown here illustrate how the arc is suspended when AV and BV vary in sign.

As well as these three items: body, fin and ogive, the program will also accept a fourth item: a known piece. That is, a piece, or any group of pieces, for which the weight, moments of inertia and center of gravity to the reference is known. This item will be included in the analysis with the other (unknown) pieces.

Output to the program is divided into five parts.

The first part is the graph of the shell. It is a true representation of all the input data and should compare exactly to the blueprint.

Figure 4 shows an example of the graphic output. This particular shell is composed of 76 body pieces and 4 fins: a total of 80 cards of input.

The scaling used in this case is 1/2 unit to the inch. Scaling is at the discretion of the user. If no scaling is specified, the best possible scaling will be used; that is, scaling which will produce a reasonably sized graph; height to diameter (X to Y direction) in the ratio of 1 to 1 and the units per inch in some workable amount as 1 unit to the inch, 2 units, 1/2 units, 1/4 units, etc.

The second part of output consists in listing all the input data, card by card, with a brief explanation of the options requested.

The third part gives the corresponding properties of each item considered independently. These properties include weight, "transfer effect" moments of inertia, center of gravity to the reference and volume. "Transfer effect" is the sum of the products of the weight and distance squared of each weight element of the item from its own center of gravity. The transfer effect is an intermediate quantity required to determine the total moment of inertia of the shell. This quantity is useful to know if revisions by hand are to be made on a shell after the computer has calculated its properties.

The fourth part of output gives the properties of the entire shell: the total weight, moments of inertia, and the center of gravity. The center of gravity, besides being printed, is also indicated on the graph of the shell, as can be seen on Figure 4.

The fifth part of output is the "Subtotal Sheet." For any piece on the subtotal sheet, the properties given are the sum of all those properties for all the preceding pieces. This feature is very useful if revisions are to be made on the shell. It enables the user to perform a sectional analysis so that alterations to any piece or group of pieces to achieve a certain total weight, moment or volume is greatly simplified.

Figure 5 is an illustration of a shell which contains ogive pieces. The data for ogive pieces is particularly error prone. Very often the center or direction, of the arc has been incorrectly determined. The graph of the ogive is usually sufficient to point out these errors.

Figure 6 is an illustration of a shell which contains an input data error. This error, occurring between heights 16 and 17, is clearly visible and eliminates the necessity of checking the almost 200 input cards needed for this run.

In order to run this program, three input cards are needed, followed by the body, fin, ogive and known items (one card per item).

The first card is used for a title. The information written on this card will appear on the output sheets and, if desired, on the graph as well.

The second card is the option control card. Here are given options governing five general areas.

1. Graph or no graph output;
2. Scaling on the graph, which has already been described;

3. Size of the graph. This option, if specified, will cause an 11 by 11 inch graph to be produced. Of course, in this case, reasonable scaling must be forsaken for size;
4. A format option. Normally, the field width for each dimension is 10. However, with this option it is possible to punch the dimensions using no field width, but instead by separating each number by a comma. Also, whole numbers need not have decimals and E-type numbers are acceptable; and
5. The last option concerns dimension change in subsequent runs.

Often, especially when a shell is in the design stage, it is important to know what happens when certain dimensions are varied, deleted or added. This option will cause the computer to hold all input values after the first run and then to pick up any deletions, changes or additions on the second, third, fourth, etc., runs.

If the option control card is left blank, the field width format is set at 10, a one unit to the inch graph will be produced and the program will consider each run independent.

The third card of input is the Index Card. Here is given the number of body, fin, known and ogive items. Also, the number of pieces each fin is sectioned into is given, as well as the total number of copies of output and subtotal sheets desired. Certain values on this card may be left blank, if desired.

For example, if the number of body pieces is not specified, the program will scan the next card for a "B" which means that the following cards are body items. The body items, in this case, will be terminated with a blank card.

If no "B" is found, the program will assume that there are no body items in the run. The same holds true for ogive, fin and known items. This feature eliminates the necessity of counting the number of pieces (and hence cards) in any one group.

In conclusion: this shell program is not particularly new to Picatinny Arsenal. It has been in use since 1964 and seems to be very useful in both designing and evaluating artillery shells. Its output is readily acceptable by other computer programs on the Arsenal such as trajectory and stability programs.

It was written in FORTRAN IV for the IBM 7090, originally, and has since been converted for use on the IBM 360, Models "40" and "65". The plotter used is CALCOMP, Model 570/563, magnetic tape. The program is fully described in a Picatinny Arsenal Technical Report, Number 3327.

## BODY ITEMS

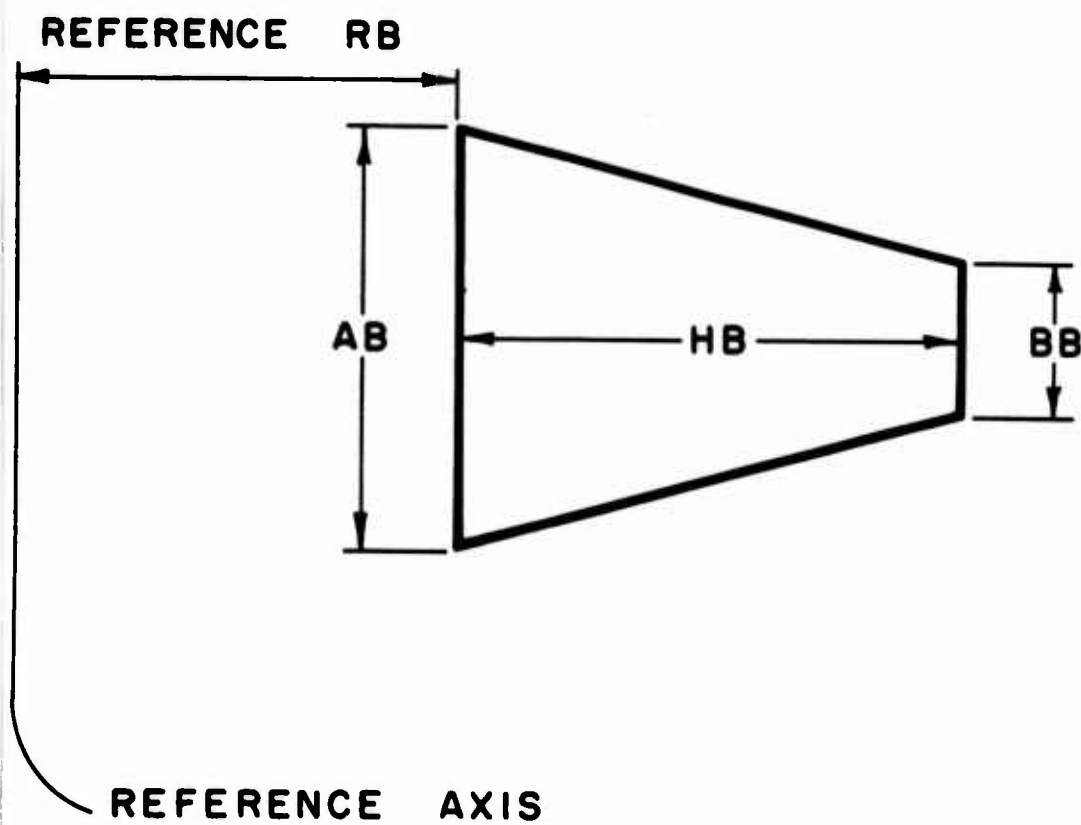


FIGURE 1

## FIN ITEMS

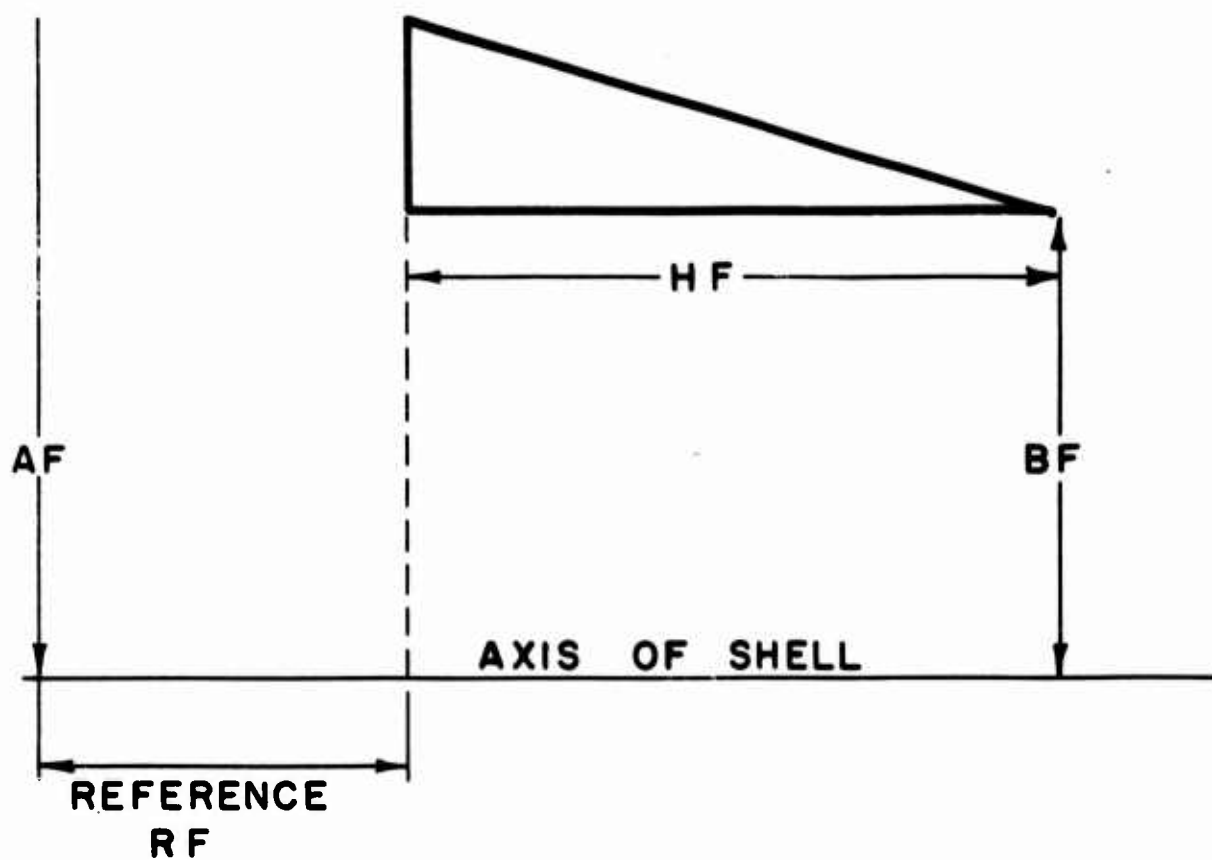


FIGURE 2

**BLANK PAGE**



# OGIVAL ITEMS

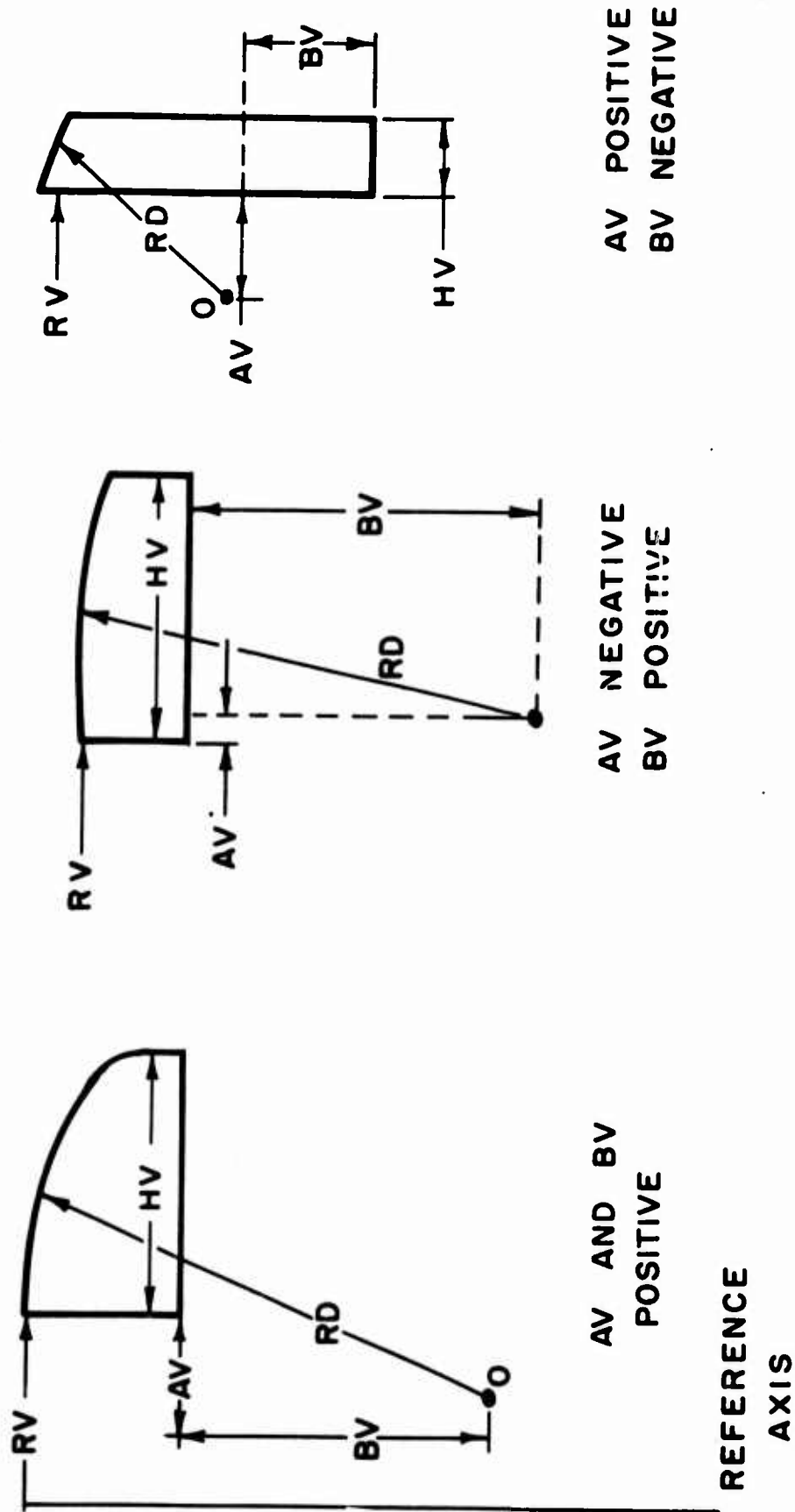
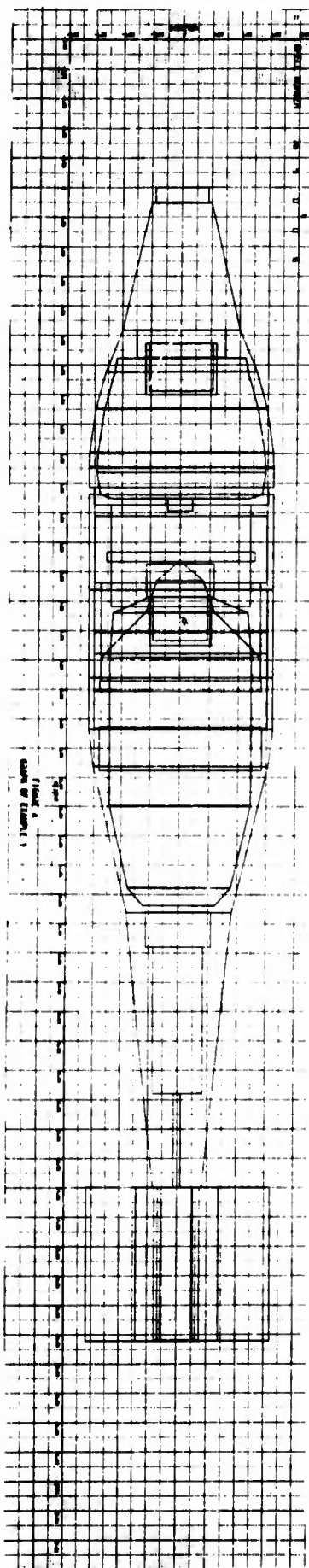
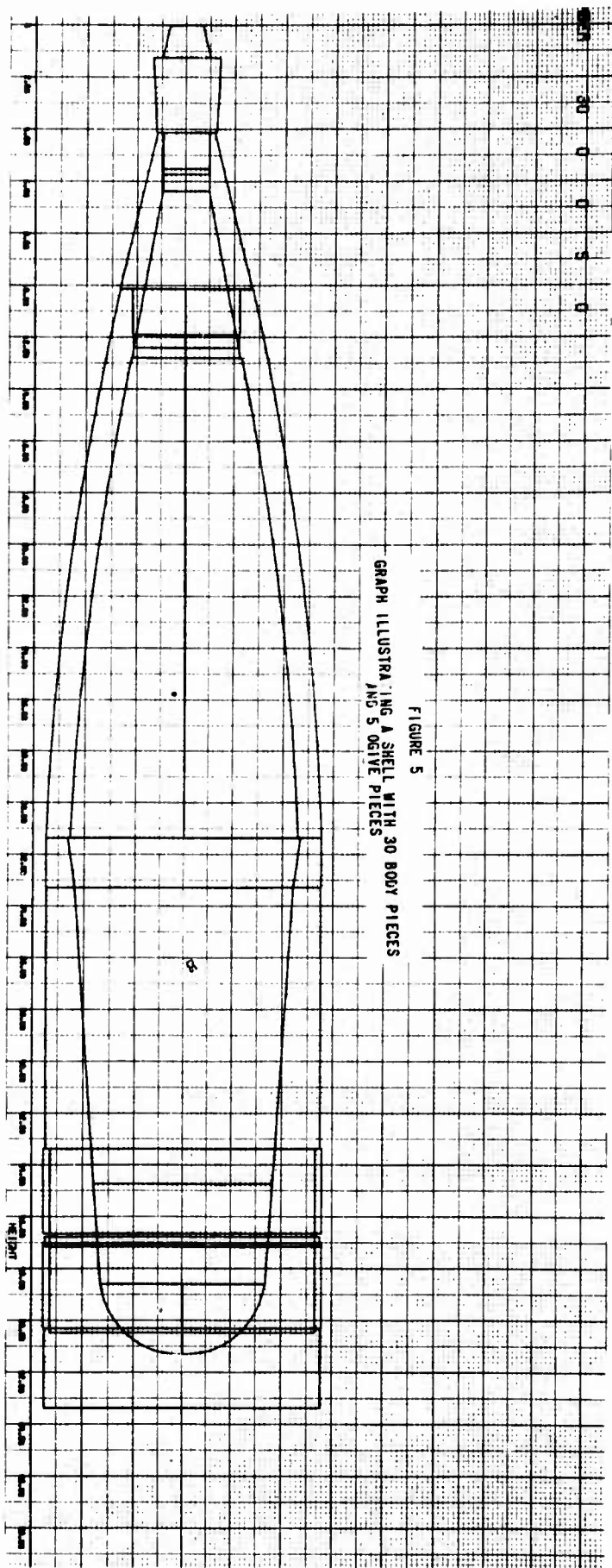
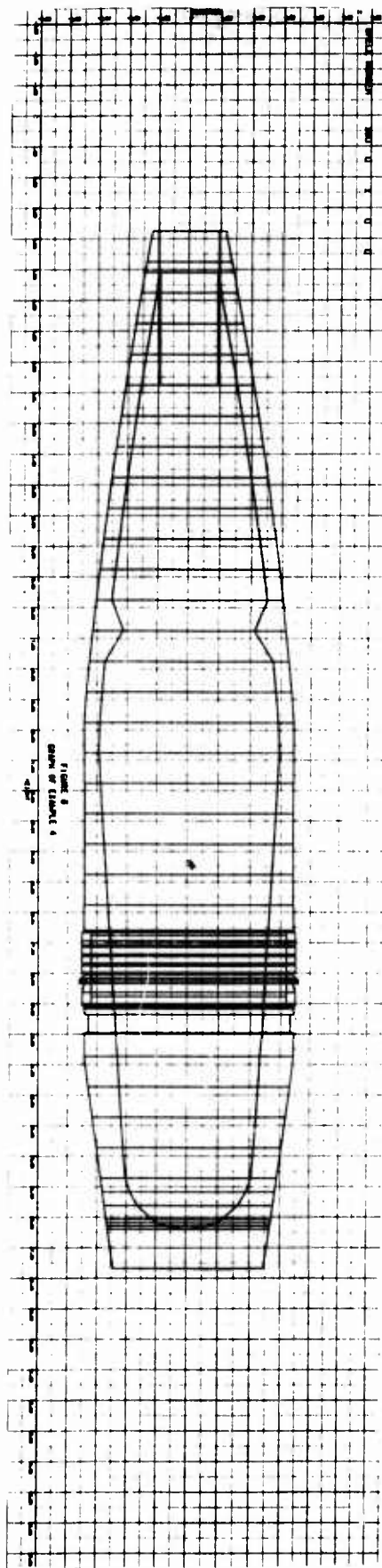


FIGURE 3







ZERPOL, A ZERO FINDING ALGORITHM FOR POLYNOMIALS USING  
LAGUERRE'S METHOD\*

Brian T. Smith  
Department of Computer Science  
University of Toronto

ABSTRACT. ZERPOL is a subroutine which computes the  $N$  zeros of the polynomial  $P(z)$  when given just its real coefficients  $A(I)$  :

$$P(z) = A(1)z^N + A(2)z^{N-1} + \dots + A(N)z + A(N+1) .$$

The zeros are stored in the complex array  $Z$  with the complex zeros appearing in complex conjugate pairs. Except for polynomials of degrees one and two, ZERPOL iterates towards a zero using Laguerre's method, which is cubically convergent for isolated zeros and linearly convergent for multiple zeros. The maximum length of the step between successive iterates is restricted so that the iterate  $x_{j+2}$  lies inside a certain region about the iterate  $x_j$  proved to contain a zero of the polynomial. An iterate is accepted as a zero when the polynomial value at that iterate is smaller than a computed bound for the rounding error in the polynomial value at that iterate. The original polynomial is deflated after each real zero or pair of complex zeros is found, and subsequent zeros are found using the deflated polynomial.

INTRODUCTION. The problem is to find the  $N$  zeros  $z_j$  of the given polynomial

$$P(z) = \sum_{j=0}^N u_j z^{N-j}$$

that satisfy

$$P(z) = u_0 \prod_{j=1}^N (z - z_j) .$$

The algorithm ZERPOL is intended to solve this problem. The algorithm is described under two sections. Section one gives a summary of the strategy used and section two describes some of the pertinent details about the implementation of this strategy in FORTRAN IV on an IBM 7094-II.

Laguerre's method is defined now: Starting with an arbitrary complex point  $x_0$ , Laguerre's method generates a sequence of iterates  $(x_j)$  for the polynomial  $P(z)$  given by

$$x_{j+1} = x_j + \mathcal{L}(x_j)$$

\*"The program Zerpole discussed in this article was developed under the direction of Professor William M. Kahan, University of Toronto, Toronto, Canada. This material was presented at the Conference by Professor Kahan who described the rationale for the program Zerpole described here by Mr. Smith." The next article in these Proceedings was submitted by Dr. Kahan and is intended to support the material in this article.

where  $\mathcal{L}(x_j)$  is the Laguerre step at  $x_j$  and equals

$$\frac{-N P(x_j)}{P'(x_j) \pm \sqrt{(N-1)^2 P'(x_j)^2 - N(N-1) P(x_j) P''(x_j)}}$$

the  $\pm$  sign being chosen so as to maximize the denominator's magnitude. (See Wilkinson (1965) for a development of Laguerre's method.)

**SUMMARY OF THE STRATEGY USED IN ZERPOL.** The overall strategy of ZERPOL is described now. Polynomials of degree  $N \leq 2$  and polynomials whose leading or trailing coefficients vanish are treated separately. The coefficients of the polynomial are scaled upward as far as possible so that spurious underflow does not occur when the polynomial is evaluated near a zero. ZERPOL first attempts to start the iterative procedure at the origin. If the origin is not an acceptable initial iterate, trial initial points in a certain annular region around the origin are tested until a suitable initial iterate is found. Subsequent iterates are restricted in order that the modulus of the polynomial decreases from one iterate to the next iterate and that the distance between successive iterates is not too large. The sequence of iterates terminates when the modulus of the polynomial becomes negligible. The polynomial is deflated by the final iterate and the iteration procedure is repeated using the deflated polynomial.

Specific details of the strategy are described now. The zeros of polynomials of degree  $N \leq 2$  are computed using the standard closed formulas. The quadratic equation solver subroutine QDRTC (A,B,C,ZS,ZL) is used to compute the complex roots ZS and ZL of any real quadratic equations

$$Az^2 + Bz + C = 0$$

that must be solved by ZERPOL. Unless over/underflow occurs, the real and complex components of ZS and ZL are computed within an accuracy of 2.25 units in their last place, and  $|ZS| < |ZL|$  within the specified accuracy of these roots. Overflow and underflow occurs only when the exact roots overflow or underflow.

For the remainder of this description we assume that the  $N + 1$  real coefficients  $u_j$  are given for the polynomial

$$P(z) = \sum_{j=0}^N u_j z^{N-j} \quad \text{so that } u_0, u_N \neq 0 \text{ and } N \geq 3.$$

(Whenever  $u_0 = 0$ , the zero  $z_N$  is set to the largest number in the machine, an overflow message is enabled and the polynomial  $P(z)$  is treated as a polynomial of degree  $N-1$ . If  $u_N = 0$ ,  $z_N$  is set to zero and the polynomial  $P(z)$  is treated as a polynomial of degree  $N-1$ .)

First, the coefficients  $u_j$  are scaled so that  $\max_{0 \leq j \leq N} |u_j| \geq 2^{101}$ .

Scaling the coefficients in this manner reduces the possibility of underflow of  $P(z)$  near a zero. However the underflow condition cannot be completely eliminated as shown by the following example:

$$2^{-128} (z^{61} - z^{60}) + 2^{126} (z^{31} - z^{30}) + 2^{-128} (z-1).$$

This polynomial cannot be evaluated near any of its zeros, namely 1 and  $2^{+127/15} e^{(2k+1)\pi i/30}$  for  $k=1,2,\dots,30$ , without using numbers that overflow  $2^{127}$  or underflow  $2^{-129}$ , the limits on the 7094-II.

Next, an annular region about the origin known to contain the smallest zero of the polynomial is computed. The radius of the inner circle is the Cauchy lower bound  $R$ , namely the positive zero of the polynomial.

$$S(z) = \sum_{j=0}^{N-1} |u_j| z^{N-j} - |u_N|.$$

The radius of the outer circle is the minimum of the geometric mean  $G =$

$|u_N/u_0|^{1/N}$  of the magnitude of the zeros, the Fejer bound  $|F|$ , the Laguerre bound  $\sqrt{N} |L|$  and the Cauchy upper bound. Details concerning the computation of these bounds will be given later.

This annular region known to contain a zero of the polynomial is used to find an acceptable initial iterate for the iterations procedure. The strategy is first to attempt to start the iteration procedure at the origin. The origin is accepted as an initial iterate whenever the Laguerre step from the origin lies within the outer circle of the annulus. Otherwise the origin is unacceptable as an initial iterate and a search of this annular region for an initial iterate is started. A trial point  $x_0$  in this annular region is accepted as an initial iterate whenever the next iterate  $x_1 = x_0 + \mathcal{L}(x_0)$  roughly lies within the annulus. The trial points lie on four equiangular spirals about the origin starting on the inner circle of this annular region.

Once a suitable initial iterate has been found, subsequent iterates are determined by the following conditions: for  $j=0, 1, \dots$

$$(1) \quad x_{j+1} = x_j + L(x_j), \text{ and}$$

$$|P(x_j)| > |P(x_{j+1})|$$

where  $L(x_j)$  may be a modified Laguerre step, and

$$(2) \quad x_{j+1} + \mathcal{L}(x_{j+1}) \text{ roughly lies inside a circular region about the iterate } x_j \text{ of radius } |F| \text{ known to contain a zero of } P(z) \text{ (i.e.}$$



$|L(x_{j+1})| \leq |F|$ ), and the modified Laguerre step

$$L(x_{j+1}) = \begin{cases} L(x_{j+1}) & \text{when } |L(x_{j+1})| \leq |F|/2 \\ |F|/2 \cdot L(x_{j+1}) / |L(x_{j+1})| & \text{when } |F|/2 < |L(x_{j+1})| \leq |F|. \end{cases}$$

The modified Laguerre step may be further modified when condition (1) is not satisfied. If  $|P(x_j + L(x_j))| \geq |P(x_j)|$  then  $L(x_j)$  is replaced by  $L(x_j)/2$  and the condition (1) is retested. This process is repeated until condition (1) is satisfied. If  $|L(x_{j+1})|$  is too large (that is,  $(|L(x_{j+1})| > |F|)$  then  $L(x_j)$  is again replaced by  $L(x_j)/2$  and conditions (1) and (2) are retested. The process is repeated until both condition (1) and (2) are satisfied. (These conditions are based on theorems due to W. Kahan. See also B.T. Smith (1967).)

The iteration procedure stops whenever the polynomial value at an iterate becomes smaller than a bound on the rounding error in the polynomial value computed at that iterate. For a real point  $X$ , we can show that a bound for the rounding error in the computed value  $Q_N$  of  $P(X)$  using the Newton-Horner recurrence is given by

$$|P(X) - Q_N| \leq \sigma c E$$

where

(1) The numbers  $Q_j$  for  $j = 0, \dots, N$  are the computed values for  $q_j$  obtained from the Newton-Horner recurrence;

$$q_0 = u_0 \text{ and}$$

$$\text{for } j = 1, \dots, N \quad q_j = u_j + q_{j-1},$$

$$(2) E = \sum_{j=0}^N |Q_j| |X|^{N-j},$$

(3)  $\sigma$  equals a unit in the last place in the arithmetic used to compute  $Q_N$ , and

(4)  $c$  is machine constant of the order of 10 for IBM-7094-II representing the roundoff errors in the arithmetic used to compute  $Q_N$ .

Since a zero of  $P(z)$  need not be representable in the machine, we really want a bound for  $|P(x) - Q_N|$  where  $x$  is in the neighborhood of  $X$ , that is  $|x - X| \leq |X| \sigma$ . We can show that whenever  $|x - X| \leq \sigma |X|$  then  $|P(x) - P(X)| \leq \sigma E$  so that

$$|P(x) - Q_N| \leq \sigma(c+1)E$$



We summarize the results of this error analysis by saying that we cannot distinguish any point  $X$  for which  $|Q_N| \leq \sigma (c+1) E$  from a zero of the polynomial  $P(z)$ , and that the machine representable numbers which are immediate neighbours of a zero  $x$  of  $P(z)$  satisfy  $|Q_N| \leq \sigma (c+1) E$ .

The numbers  $q_j$  are the coefficients of the quotient polynomial  $Q(z)$  in division of  $P(z)$  by the factor  $z-X$ . That is,

$$\begin{aligned} P(z) &= \sum_{j=0}^N u_j z^{N-j} \\ &= (z-X) \sum_{j=0}^{N-1} q_j z^{N-1-j} + q_N \\ &= (z-X) Q(z) + q_N. \end{aligned}$$

Therefore the first derivative of  $P(z)$  at  $X$  can be obtained by applying the Newton-Horner recurrence to coefficients  $q_j$ .

Thus

$$\begin{aligned} Q(z) &= (z-X) W(z) + w_{N-1} \text{ and} \\ P'(X) &= w_{N-1}. \end{aligned}$$

Similarly for the second derivative  $P''(x)$ ,

$$\begin{aligned} W(z) &= (z-X) V(z) + v_{N-2} \text{ and} \\ P''(z) &= v_{N-2}. \end{aligned}$$

Notice that the error bound  $E$ , the polynomial value and its derivatives can all be computed within the same loop.

The evaluation of the polynomial value, its derivatives and the error bound  $E$  at a complex iterate  $Z$  is obtained in a similar manner to the real iterate  $X$  by replacing each occurrence of the linear factor  $(z-X)$  by the real quadratic factor  $(z-Z)(z-\bar{Z})$  where  $\bar{Z}$  is the complex conjugate of  $Z$ . (This evaluation procedure for complex points appears in Wilkinson (1965), page 447-449.)

Once an iterate is accepted as a zero, the coefficients  $q_j$  of the quotient polynomial  $Q(z)$  replace the coefficients  $u_j$  and the iteration process is repeated on the deflated polynomial. Purification of the zeros is not attempted by ZERPOL.

PROGRAMMING DETAILS FOR ZERPOL STORAGE ALLOCATION. The coefficients of the polynomial are transferred to the double precision array DU. This array DU is placed in COMMON with library workspace LIBWSP so that the workspace need not be supplied by the user. This places a restriction  $N \leq 79$  on the maximum degree of the polynomial handled by ZERPOL. However, the restriction can readily be eliminated by increasing the dimension of library workspace in the

calling program to at least  $2N + 2$  where  $N$  is the degree of polynomial. Notice that the double precision leading coefficient DU0 can be referenced DU(10) where  $10 = 0$ .

The complex array Z of zeros is treated inside ZERPOL as a double precision array so that those elements of the array Z which do not contain zeros of the polynomial may be used to store temporarily the coefficients of the quotient polynomial. The coefficients of the quotient polynomial are transferred to to DU array whenever an iterate is accepted as a zero.

All diagnostic messages initiated from ZERPOL appear in DATA statements and are issued through the subroutine UNCLE.

INITIALIZATION. The function subroutine C112(N) converts the integer  $N$  from its binary representation to its binary coded data (BCD) representation in order to appear in the diagnostic messages given by UNCLE.

A warning message is generated when  $N > 79$ . However this message is suppressed for all subsequent calls of ZERPOL in the same job.

Over/underflow variables OVFLOW and UNFLOW are saved from the user's program. The statement NSAV=NFPTST(0) suppresses any messages for the over/underflow occurring in ZERPOL. (See Programmer's Reference Manual (PRM), (1964).)

SCALING. If  $N \leq 2$ , or  $\max_{0 \leq J \leq N} |DU(J)| > 2^{101}$  the coefficients are not scaled. Otherwise the coefficients are scaled by the scale factor DSC so that the  $\max_{0 \leq J \leq N} |DU(J)| = 2^{101}$ . The scaling procedure is executed in the unnormalized mode (CALL FPTUN) in order to extend the allowed lower limit to the magnitude of the coefficients. As a result, ZERPOL can confidently ignore underflow except when underflow occurs in the first and last coefficients. The standard mode is re-instated with CALL FPTST.

Overflow may occur in the evaluation of the polynomial and its derivatives. When overflow does occur, we attempt to remove the overflow condition by scaling down the coefficients by  $2^{-27}$ .

If the leading coefficient becomes unnormalized in the process of scaling down the coefficients, a message is given stating that the polynomial cannot be evaluated near some of its zeros without over/underflow.

THE ANNULUS CONTAINING THE SMALLEST ZERO,  $R \leq |z| \leq G'$ . The geometric mean  $G$  of the magnitudes of the zeros is computed using logarithms in order to prevent over/underflow of the intermediate results.

The reciprocal of the Newton step at the origin is checked for overflow. If it overflows, a zero is close enough to the origin to be considered as zero. Also  $|P'(0)/P(0)| < 2^{+127}$  ensures that the Cauchy lower bound  $R$  doesn't underflow.

The Fejer bound at the point X is the magnitude of the zero F of smaller magnitude of the quadratic equation

$$(P''(X)/(2N(N-1)))F^2 + (P'(X)/N)F + P(X)/2 = 0.$$

The Laguerre step  $\mathcal{L}(X)$  is simply related to the zero F by the formula

$$\mathcal{L}(X) = F / ((N-2) P'(X) F / (N P(X)) + N-1) .$$

The values of the polynomial and its two derivatives at the origin are given by the coefficients (DU(N) , DU(N-1) and DU(N-2)).2 . The Fejer bound is computed using the subroutine QDRTC and the Laguerre step and Laguerre bound at the origin are computed immediately. Thus

$$B = 1.0001 \min ( \sqrt{N} \mathcal{L}(0), |F| , G )$$

is an upper bound for the magnitude of the smallest zero of the polynomial.

Next, the Cauchy lower bound R for the smallest zero is computed where R is the positive zero of the polynomial

$$S(z) = \sum_{I=0}^{N-1} |DU(I)| z^{N-I} - |DU(N)| .$$

This zero R can readily be computed using the Newton-Raphson method with  $x_0 = B$  because all the derivatives of S(z) are positive for z positive.

Notice that for  $X \geq R$

$$X S'(X) \geq |DU(N)| ,$$

so

$$S'(X) \geq 2^{-129} .$$

Therefore we do not expect  $S'(X)$  to underflow. The sequence of iterates  $(x_j)$  terminates when  $x_{j+1} \geq x_j$  for the first time. If overflow has occurred in the computation of the last iterate, that iterate is probably incorrect and can be corrected easily only by scaling down the coefficients of the polynomial. If no overflow occurs,  $0.99999 x_j$  is accepted as a lower bound for the magnitude of the smallest zero of the polynomial. Since  $R/(2^{1/N}-1)$  is an upper bound for the smallest zero of P(z) and  $R/(2^{1/N}-1) \leq N(1.445) R$ , then  $G' = \min (B, N(1.445)R)$  is accepted as an upper bound for the magnitude of the smallest zero of the polynomial P(z).

THE ITERATION PROCEDURE. The strategy of this section of the algorithm has been described previously in section one. To assist the reader in following the FORTRAN code, STARTD and SPIRAL are logical variables indicating whether or not the iteration procedure has started successfully and whether or not a spiral search for an initial iterate has started.

Laguerre's method may be exact for zeros of multiplicity N-1 and N

so that the initial iterate from the origin is allowed to reach the outer circle of the annulus  $R \leq |z| \leq G'$  whenever this annulus is relatively narrow (i.e.  $R > G'/2$ ).

The time required to compute the value of the polynomial and its derivatives at a real point is less than the time at a complex point so that an iterate is forced to be real whenever the imaginary part of the iterate  $x_j$  is less than one-fifth of the step  $x_j - x_{j-1}$  to that iterate.

POLYNOMIAL EVALUATIONS. The polynomial value and its first derivative are computed using double precision arithmetic while the second derivative is computed with single precision arithmetic. We felt that the improved convergence to rare multiple zeros was not worth the cost in extra time of computing the second derivative with double precision arithmetic. The unnormalized mode is used for the above computation.

The evaluations of the polynomial and its derivatives at a real iterate and at a complex iterate are done in separate blocks. The computation in the case of a real iterate is straightforward. However, precautions need be taken when the magnitude of a complex iterate is extremely large or small.

In the case of a complex iterate  $X$ , the squared modulus of the complex iterate appears in the quadratic factor and so may overflow/underflow. Thus whenever

$$\begin{aligned} |X| &\geq 2^{63.5}, \text{ (square root of overflow)} \\ \text{or} \quad |X| &< 2^{-63.5}, \text{ (square root of underflow)} \end{aligned}$$

then the coefficients of the quadratic factor  $(z-X)(z-\bar{X})$  are carefully scaled so that the possibility of overflow or underflow in the evaluation loop is minimized.

If overflow cannot be avoided in the evaluation loops the coefficients are scaled down by  $2^{-27}$ .

If the modulus of the polynomial is greater than the error bound in the computed value of the polynomial, and the modulus of the polynomial underflows, then a message is given declaring that overflow/underflow occurs in the evaluation of the polynomial near one of its zeros. The last iterate is accepted as a zero of the polynomial.

If the reciprocal of the Newton step at the last iterate overflows, then the last iterate is within a distance of  $N 2^{-127}$  of a zero of the polynomial. The last iterate is accepted as a zero of the polynomial but underflow is signalled.

SEARCHING THE ANNULAR REGION FOR AN INITIAL ITERATE. The search for an acceptable starting point for the iteration procedure starts with a point on the inner circle of the annulus in the direction of the Laguerre

step from the origin. Subsequent trial points lie on the spirals traced out by  $R(i - 1.25N)^k$  for  $k = 0, 1, \dots$  where the angle between successive trial points is  $-\tan^{-1}(N/1.25)$  or just more than  $-90^\circ$ . If every fourth trial point is examined, the locus is a spiral progressing in a counter-clockwise direction. The constant 1.25 is chosen in the hope that the distribution of the trial points is dense enough in the annular region to find an initial iterate but not so dense that a great deal of time is spent searching for a suitable initial iterate.

**TEST RESULTS.** ZERPOL was tested with polynomials given in papers by P. Henrici and B.O. Watkins (1964) and E.H. Bareiss (1965). In all cases ZERPOL satisfied our criterion for the accuracy of the zeros, namely that the coefficients of the polynomial reconstructed from the zeros given by ZERPOL closely resemble the original coefficients.

ZERPOL computes all zeros of a polynomial of degree  $N$  in roughly  $N^2$  milliseconds on our IBM-7094-II and consists of approximately 550 cards. We compared ZERPOL with the package of subroutines catalogued in 1965 as SDA-3332 in the SHARE library. This routine found the zeros of the test polynomials taking from two to five times longer than ZERPOL. We also compared results from ZERPOL with the subroutine POLRT from the IBM System/360 Scientific Subroutine Package (1966). This subroutine is about as fast as ZERPOL, but sometimes gives wrong answers.

The following table gives some statistics on the number of steps required to find all the zeros of polynomials of varying degrees. The coefficients of these polynomials are random numbers taken from a normal distribution with mean 0 and variance 1.

Degree	No. of Polynomials	Laguerre steps per iterated zero		Search steps per iterated zero		Half steps per iterated zero	
		Average	Maximum	Average	Maximum	Average	Maximum
3	100	3.9	6.0	0.3	2.0	0.02	5.0
6	28	4.5	6.0	0.54	3.7	0.14	4.0
12	7	4.7	5.2	0.55	4.3	0.25	1.7
18	4	5.0	5.5	1.20	4.3	0.46	2.0

This version of ZERPOL was produced during the author's work for an M.Sc degree at the University of Toronto under the supervision of W. Kahan, with the support of a Province of Ontario Fellowship.

#### REFERENCES

1. Bareiss, E.H. (1965), RSSR Routine, A Root-Squaring and Subresultant Procedure for Finding Zeros of Real Polynomials, ANL-6987, AEC Research and Development Report Program Material Laboratory, Illinois.
2. Henrici, P. and Watkins, B.O. (1965), Finding Zeros of a Polynomial by the Q-D algorithm, Communications of the Association for Computing Machinery, Volume 8, Number 9 (1965), pp 370-574.
3. IBM System/360 Scientific Subroutine Package (1966) Programmer's Manual, 360 A-CM-03X.
4. Programmer's Reference Manual (1964) for the IBM-7094-II Computer (2nd edition) by the staff of the Institute of Computer Science, The University of Toronto, Toronto, Canada.
5. Smith, B.T. (1967), A Zero Finding Algorithm Using Laguerre's Method, M.Sc. Thesis, University of Toronto, Toronto, Canada.
6. Wilkinson, J.H. (1963), Rounding Errors in Algebraic Processes, Her Majesty's Stationery Office, London, Chapter 2.
7. ——— (1965), The Algebraic Eigenvalue Problem Clarendon Press, Oxford, pp 443-445, 447-449.

# APPENDIX - ZERPOL LISTING

```

SUBROUTINE ZERPOL (A,NDEG,Z)
C  CALL ZERPOL(A,N,Z)  TO SET  Z(I) = I-TH ZERO OF THE POLYNOMIAL
C      A(1)*Z**N + A(2)*Z**(N-1) + ... + A(N)*Z + A(N+1) .
C  NOTE THAT  N = DEGREE OF THE POLYNOMIAL, AND THERE ARE  N+1  REAL
C  COEFFICIENTS  A(I).  DIMENSIONS -  REAL  A( AT LEAST N+1 )
C                                          COMPLEX  Z( AT LEAST N )
C  NORMALLY,  N  SHOULD NOT EXCEED 79. OTHERWISE THE PROGRAMMER SHOULD
C  INCLUDE COMMON /LIBWSP/ LIBWSP( AT LEAST 2*N+2 ) .
C
C  USE POLMY TO CHECK ACCURACY .
C
C  COMPLEX CONJUGATE ZEROS  Z(I)  OCCUR CONSECUTIVELY , I.E.
C  IF( Z(I) IS COMPLEX )  EITHER  Z(I+1) = CONJG(Z(I))
C                          OR ELSE  Z(I-1) = CONJG(Z(I)) .
C  IF ALL COEFFICIENTS  A(I)=0 , THE 0.0/0.0 DIAGNOSTIC IS PRODUCED.
C
C  *****
C
C 000 CONTINUE
C  REAL  A(80)
C  COMPLEX          Z(79)
C  DOUBLE PRECISION Z(79)
C
C  DU(I) IS THE COEFFICIENT OF  Z**(N-I)  IN THE CURRENT POLYNOMIAL .
C  DOUBLE PRECISION          DUO,          DU(79)
C  COMMON                    /LIBWSP/LIBWSP(160)
C  EQUIVALENCE                (LIBWSP , DUO ), ( LIBWSP(3), DU )
C
C  LOGICAL  OVFL, UNFL, SAVI, SAVJ, STARTD, SPIRAL
C  LOGICAL  OVFLOW , UNFLOW , TOOBIG
C  COMMON  /OVFLOW/OVFLOW, /UNFLOW/UNFLOW
C  DATA          TOOBIG/.TRUE./
C
C  DIMENSION  ACF1(2),    ACF2(2) ,    ACF(2)
C  COMPLEX    CF1,        CF2,        CF
C  EQUIVALENCE (CF1,ACF1), (CF2,ACF2), (CF,ACF)
C
C  COMPLEX    CDIR,    CSPIR
C  DIMENSION  ACDIR(2),    AC(2),    ACL(2)
C  COMPLEX    CDIR,    C,    CL
C  EQUIVALENCE (CDIR,ACDIR), (C,AC), (CL,ACL)
C
C  DOUBLE PRECISION  DZNR, DZNI, DZOR, DZOI
C  DOUBLE PRECISION  DX,    DR,    DSC,    DY,    DX2,    DV
C  EQUIVALENCE      (DX,X), (DR,R), (DSC,SC), (DY,Y), (DX2,X2), (DV,V)
C  DOUBLE PRECISION  DT ,    DT1
C  EQUIVALENCE      (DT,I) , (DT1,T1)
C  DATA          DSC/0.00/

```

```

DIMENSION MESSH(10)
DATA MESSH(1)/ 54HOA POLYNOMIAL OF DEGREE
SERUS. /, MESSH(10) / 077777777777/
COMPLEX CMESH
EQUIVALENCE (CMESH, MESSH(5))
C
DIMENSION MESH(22)
DATA MESH(1) /126HOTHERE IS SOME REASON TO BELIEVE THAT THE FIRST
$ ZERUS ARE INCORRECT. QUICKLY CALL W. KAHAN OR B.
ST. SMITH. /, MESH(22)/ 077777777777 /
COMPLEX CMESH
EQUIVALENCE (CMESH, MESH(9))
C
DIMENSION MESS(18)
DATA MESS(18) / 077777777777 /, MESS(1) /
$102HOYOUR N = EXCEEDS 79 , AND REQUIRES THE DIMEN
$SION OF LIBWSP TO BE AT LEAST 2*N+2 . /
COMPLEX CMESH
EQUIVALENCE (CMESH, MESS(3)), (FINITY, MESS(18))
C
DIMENSION UVFUNK(16)
DATA UVFUNK(16)/077777777777/, UVFUNK(1)/
$90HOZERPOL CANNOT EVALUATE THE GIVEN POLYNOMIAL NEAR SOME OF ITS Z
$ERUS WITHOUT OVER/UNDERFLOW /
C
DATA BIT /0400000000/, TM27 /014540000000/, IO/0/,
$ T635/0300552023625/, T101 /034640000000/,
$ TM645/0100552023625/
C
BIT=2.**-129=SMALLEST NU. T101=2.**101 TM27=2.**-27
C FINITY=-2.**127=-LARGEST NU. TM645=2**(-64.5)
C T635=2**(63.5)
C
SPECIAL FUNCTIONS-
C ALUG2(X) = LOGARITHM OF X TO THE BASE 2.
C TWOXP(X) = 2.**X
C CI12(J) = ALPHABETIC REPRESENTATION OF J IN I12 FORMAT (CMPLX)
C AND(X,Y) LOGICALLY 'ANDS' X AND Y BIT BY BIT .
C AMAXA(I,J,K,L) FINDS THE MAXIMUM OF ABS(T(I)) FOR I FROM J UP
C TO K IN STEPS OF L.
C AMIN1(X,Y,....,Z) FINDS THE MINIMUM OF ITS ARGUMENTS X,Y,....,Z .
C
GAMA, THETA, AND PHI ARE TEST PARAMETERS FOR ZERPUL .
DATA GAMA/0.5/, THETA/1.0/, PHI/0.2/
C UN40=40.*2**(-53) UN10=10.*2**(-53)
DATA UN40/0121500000000/, UN10/0117500000000/
C
*****

```



```

50 CONTINUE
  N = NDEG
  CMESHH = C112(N)
  IF( N .LE. 0 ) CALL UNCLE( 0, MESHH )
  IF ( .NOT. TOOBIG .OR. N .LE. 79 ) GO TO 51
    TOOBIG = .FALSE.
    CMESS = CMESHH
    CALL UNCLE( -75, MESS )
C   SAVE OVERFLOW/UNDERFLOW INDICATORS OF THE CALLING PROGRAM .
51 SAVD = OVFLOW
  SAVU = UNFLOW
  NSAVE = NFPTST(0)
  DVF = .FALSE.
  UNF = .FALSE.
C
C   MOVE THE COEFFICIENTS A(I) TO DU(I-1) .
  DO 52 I = 10, N
52   DU(I) = A(I+1)
C
C   SCALING ( ONLY WHEN N .GT. 2 )
100 CONTINUE
  IF ( N .LE. 2 ) GO TO 204
  ASSIGN 400 TO LSW
C   ( SEE STATEMENT 300 . )
  SC = AMAXA( DU0, 1, 1, 2*N+1, 2 )
  IF ( SC .EQ. 0. ) GO TO 206
  IF ( SC .GE. T101 ) GO TO 105
  SC = T101/SC
C   SCALE BY SC TO HAVE MAX(DU(I), I=0, N) APPROACH 2.**100 .
  GO TO 103
C
C   ( RE-SCALING NECESSITATED BY OVERFLOW USES SC=2.**(-27) . )
102 SC = TM27
C
103 CONTINUE
  CALL FPTON
  DO 104 I = 10, N
104  DU(I) = USC*DU(I)
  CALL FPTST
C   FIND NUMBER I OF CONSECUTIVE LEADING COEFFICIENTS EQUAL TO ZERO .
105 DO 106 I = 10, N
  IF ( AND( DU(I), BIT ) .NE. 0. ) GO TO 107
C   EACH VANISHED LEADING COEFFICIENT YIELDS AN INFINITE ZERO .
  J = N-I
106  Z(J) = FINITY
107 IF ( I .EQ. 0 ) GO TO 204
C
C   SLIDE BACK COEFFICIENTS AND DECLARE OVERFLOW .
  DO 108 K = I, N
108  DU(J) = DU(K)

```

```

      N = N-1
      IF ( SC .EQ. TM27 ) CALL UNCLE( 73, UVF, UNF )
      UVF = .TRUE.
      GO TO 203
C
C RE-ENTRIES FOR CURRENT (REDUCED) POLYNOMIAL .
201 N = N1
202 N = N-1
203 ASSIGN 400 TO LSW
C ( SEE STATEMENT 300 .)
204 OVFLOW = .FALSE.
      UNFLOW = .FALSE.
      IF ( N-2 ) 205, 206, 300
205 Z(1) = DSIC( CMPLX( RND( -DU(1)/DU0 ) , 0. ) )
      GO TO 207
C
206 CALL QDRTC( RND(DU0), RND(DU(1))+0., RND(DU(2))+0., Z(2), Z(1) )
207 UVF = UVF.OR.OVFLOW
      UNF = UNF.OR.UNFLOW
C RESTORE OVERFLOW AND UNDERFLOW INDICATORS AND ENABLE MESSAGE .
      OVFLOW = SAVO
      UNFLOW = SAVO
      NSAVE = INPTST(NSAVE)
C PROVIDE ONLY THE RELEVANT OVER/UNDERFLOW MESSAGES.
      IF(OVF) SC = FINITY*FINITY
      IF(UNF) SC = BIT*BIT
      RETURN
C
C CHECK FOR ZEROS = (0.,0.) (HENCEFORTH N.GT. 2 )
300 IF ( AND( DU(N) , BIT ) .NE. 0. ) GO TO LSW,( 400, 700 )
C (ENTRY FROM BLOCK 500 IF RECIPROCAL OF NEWTON STEP OVERFLOWS .)
301 IF ( SNGL(DU(N)) .NE. 0. ) UNF = .TRUE.
302 Z(N) = 0.00
      GO TO 202
C
C HENCEFORTH N.GT. 2. DU0 .NE. 0. . AND DU(N) .NE. 0.
C INITIALIZE SOME USEFUL CONSTANTS.
400 CONTINUE
      XN = N
      XN1 = XN - 1.
      XN2 = XN1 - 1.
      X2N = 2./XN
      X2N1 = X2N/XN1
      XN2N = XN2/XN
      N1 = N-1
      RTN = SORT(XN)

```

*Note: RND(D) rounds double-precision D to single precision.*  
*QDRTC(A,B,C,Zs,ZL) solves  $AZ^2+BZ+C=0$  for Zs and ZL.*

```

C
C   CALCULATE G , AN UPPER BOUND FOR THE NEAREST ZERO .
C   START WITH G = CABS( GEOMETRIC MEAN OF THE ZEROS ) .
500 G = TWOXP( (ALOG2(ABS(DU(N))) - ALOG2(ABS(DU0))) /XN + 1.E-5 )
C
C   CALCULATE LAGUERRE-STEP CDIR AND FEJER-BOUND FOR G .
C   CALCULATION OF THE LAGUERRE STEP INVOLVES THE SQUARE OF
C   RECIPROCAL OF NEWTON'S STEP. SINCE IT CAN EASILY OVERFLOW, THE
C   FEJER BOUND IS CALCULATED WITH NO SUCH OVERFLOWS AND THE
C   LAGUERRE STEP IS CALCULATED FROM IT.
OVFLOW = .FALSE.
R = SNGL( DU(N-1) )/SNGL( DU(N) )
C   IF OVFLOW, A ROOT OF POLY. IS WITHIN N*2**(-127) OF 0.
IF ( OVFLOW ) GO TO 301
CALL QDRTC( X2N1*SNGL(DU(N-2)) , X2N*SNGL(DU(N-1)) , SNGL(DU(N)) ,
$   C , CF1 )
R = XN2N*R
CDIRO = C/CMPLX( R*AC(1) + XN1 , R*AC(2) )
ABDIRO = ABS(REAL(CDIRO)) + ABS(AIMAG(CDIRO))
G = AMIN1( G , 1.0001*AMIN1(ABS(AC(1)) + ABS(AC(2)) , RTN*ABDIRO ) )
C
C   CALCULATE THE CAUCHY-LOWER BOUND R FOR THE SMALLEST ZERO BY
C   SOLVING ABS(DU(N)) = SUM( ABS(DU(I))*R**(N-I) , I = 0, N-1 )
C   USING NEWTON'S METHOD .
R = G
CALL FPTUN
601 T = ABS(DU0)
S = 0.
OVFLOW = .FALSE.
DO 602 I = 2,N
S = R*S + T
602 T = R*T + ABS( DU(I-1) )
S = R*S + T
C   IT CAN BE PROVED THAT S CANNOT UNDERFLOW .
T = (R*T - ABS( DU(N) ) )/S
S = R
R = RND( R - T )
IF ( R.LT. S ) GO TO 601
IF ( OVFLOW ) GO TO 102
C
C   R/(2**(1/N) - 1) .LT. 1.445*N*R IS ANOTHER UPPER BOUND , SO SET
GU = AMIN1( 1.445*XN*R , G )
RU = 0.99999*S
ASSIGN 700 TO LSW
C   ( SEE STATEMENT 300 . UNLESS DEGREE OF POLY. IS REDUCED, RO, GO
C   AND ABDIRO ARE UNCHANGED .
C   NOW RO .LT. CABS( SMALLEST ZERO ) .LT. GU
C

```

```

C      INITIALIZE THE ITERATION TO BEGIN AT THE ORIGIN .
700 CONTINUE
   FEJER = GU
   G = GU
   CDIR = CDIRU
   ABDIR = ABDIRU
   DZNR = 0.00
   DZNI = 0.00
   FN = ABS(DU(N))
   SPIRAL = .FALSE.
   STARTU = .FALSE.

C
C      RE-ENTRY POINT TO ACCEPT , MODIFY, OR REJECT THE LAGUERRE STEP .
C      GAMA, THETA, PHI ARE ARBITRARY PARAMETERS . ZERPUL IS TO BE TESTED
C      FOR SPEED AND RELIABILITY WHEN THEY ARE VARIED. POSSIBLE
C      VALUES ARE GAMA=0.5, THETA=1.0, PHI=0.2 .
701 V = ABDIR/G
C      ACCEPT CDIR IF CABS(CDIR) .LE. GAMA*G .
      IF( V .LE. GAMA ) GO TO 800
C      REJECT CDIR IF CABS(CDIR) .GT. THETA*G .
      IF ( V .GT. THETA ) GO TO 1100
C      MODIFY CDIR SU THAT CABS(CDIR) = GAMA*G .
      IF( .NOT.( STARTU .OR. SPIRAL ) .AND. RU .GT. GAMA*G ) GO TO
          5      800
      V = GAMA/V
      CDIR = CMPLX( V*ACDIR(1) , V*ACDIR(2) )
      ABDIR = ABDIR*V

C
C      ACCEPT PREVIOUS ITERATE. SAVE DATA ASSOCIATED WITH CURRENT ITERATE
800 CONTINUE
   G = FEJER
   CL = CDIR
   ABSCL = ABDIR
   FO = FN
   DZOR = DZNR
   DZOI = DZNI
C      CDIR AT THE ORIGIN IS IN THE DIRECTION OF DECREASING FUNCTION
C      VALUE SO
   STARTU = .TRUE.
C      THE NEXT ITERATE IS ZN=CMPLX( DZNR , DZNI ), WHERE
C      (ENTRY POINT WHEN CDIR IS NOT ACCEPTED. )
801 DZNR = DZOR + ACL(1)
   DZNI = DZOI + ACL(2)
C      IS ZN CLOSE TO THE REAL AXIS RELATIVE TO STEP SIZE .
C      (ENTRY POINT FROM THE SPIRAL BLOCK.)
802 IF ( ABS(DZNI) .LE. PHI*ABSCL ) GO TO 950

```

```

900 CONTINUE
C   ZN IS COMPLEX .
C   FACTORIZATION OF POLYNOMIAL BY QUADRATIC FACTOR (Z**2-X2*Z+R)
C
C   SUM(DU(I)*Z**(N-I)) = (Z**2-X2*Z+R)*SUM(Z(I)*Z**(N-I-2)) +
C                       Z(N-1)*(Z-X) + Z(N)   FOR ALL Z ,
C   THE VALUE OF THE POLYNOMIAL AT (X,Y) IS CF .
C   FIRST DERIVATIVE OF POLYNOMIAL AT (X,Y) IS CF1 , AND
C   SECOND DERIVATIVE OF POLYNOMIAL AT (X,Y) IS 2.*CF2 ,
C   WHERE (X,Y) IS A ZERO OF Z**2-X2*Z+R .
C   E IS ERROR BOUND FOR THE VALUE OF POLYNOMIAL AND
C   Z(I) ARE THE COEFFICIENTS OF QUOTIENT POLYNOMIAL .
C   BE SURE THAT THE OVERFLOW INDICATOR IS TURNED OFF.
OVFLOW = .FALSE.
C   CALL FPTUN TO REDUCE ERRORS CAUSED BY INTERMEDIATE UNDERFLOWS .
CALL FPTUN
C   INITIALIZATIONS FOR THE EVALUATION LOOPS .
901 S = 0.
   S1 = 0.
   T1 = 0.
   DT = DUO
C   INDEX J IS USED TO CHANGE DX ON THE LAST ITERATION .
   J = 3
C   SET Z(X,Y) TO ZN(ZNR,ZNI) .
   DX = DZNR
   DY = DZNI
C   SC IS ESTIMATED IN CASE SCALING IS NEEDED IN BLOCK 900 .
   SC = CABS( CMPLX(DX,DY) )
C   IF CABS(ZN) .LE. SQRT( SMALLEST NO. ) , SCALE UP X AND Y .
C   IF CABS(ZN) .GE. SQRT( LARGEST NO. ) , SCALE DOWN X AND Y .
   IF( SC .GE. T635 .OR. SC .LE. T645 ) GO TO 905
C   SCALING OF X2 AND R IS UNNECESSARY.
   DX2 = DX + DX
   DR = DX**2 + DY**2
   Z(1) = DU(1) + DX2*DUO
   Z(2) = DU(2) + ( DX2*Z(1) - DR*DUO )
   IF ( J .LT. N ) GO TO 903
902   DX2 = DX
      J = N
903   DO 904   I = J,N1
      V = S1*R
      S1 = S
      S = T1 + (X2*S - V )
      DV = DT1*DR
      DT1 = DT
      DT = (DX2*DT - DV ) + Z(I-2)
904   Z(I) = DU(I) + ( DX2*Z(I-1) - DR*Z(I-2) )
      IF ( J .LT. N ) GO TO 902
      GO TO 909

```

```

C
C 905 SCALE X AND Y LEST R OVERFLOWS OR UNDERFLOWS .
      DX = DX/DSC
      DY = DY/DSC
C      DR CANNOT OVERFLOW, FORTUNATELY .
      DR = ( DX**2 + DY**2 ) * DSC
      DX2 = DX + DX
      Z(1) = DU(1) + (DX2*DU0)*DSC
      Z(2) = DU(2) + ( DX2*Z(1) - DR*DU0 ) * DSC
      IF( J .LT. N ) GO TO 907
906      DX2 = DX
      J = N
907      DO 908 I = J, N1
          V = S1 * R
          S1 = S
          S = T1 + (X2*S - V)*DSC
          DV = DT1 * DR
          DT1 = DT
          DV = DX2 * DT - DV
          DT = Z(I-2) + DV * DSC
908      Z(I) = DU(I) + ( DX2*Z(I-1) - DR*Z(I-2) ) * DSC
      IF( J .LT. N ) GO TO 906
C
C (ENTRY POINT FROM THE NON-SCALING BLOCK . )
909 CF = CMPLX( Z(N) , DZNI*Z(N-1) )
      FN = CABS(CF)
C      IF OVFLOW, THE COEFFICIENTS MUST BE SCALED DOWN.
      IF(OVFLOW) GO TO 102
      E = ABS(DU0)
      DO 910 I = 1, N
910      E = ABS(Z(I)) + SC * E
      E = UN40 * E
      IF(OVFLOW) E = XN * E
C      CHECK TO SEE IF ZN IS A ZERO.
      IF( FN .LE. E ) GO TO 1001
C      IF FN HAS UNDERFLOWED, GIVE THE MESSAGE OVFLUNF .
      IF( AND( BIT , FN ) .NE. 0. ) GO TO 911
          CALL UNCFE( 73, OVFLUNF )
          GO TO 1000
911 CALL FPTST
C      HAS THE FUNCTION VALUE DECREASED .
      IF( FN .GE. FO .AND. STARTD ) GO TO 1100
C
      DV = 2.00 * DZNI
      CF1 = CMPLX( RND( Z(N-1) - (DV*(DT1*DZNI)) , RND( V*T ) )
      CF2 = CMPLX( T - V*(V*S) , SNGL(DZNI)*(3.*T1 - V*(V*S1)) )
C      FIND THE LAGUERRE STEP AT ZN.
      OVFLOW = .FALSE.
      C = CF1/CF
C      IF OVFLOW, THERE IS A ZERO WITHIN A DISTANCE OF N*2**(-127) OF ZN
      IF(OVFLOW) GO TO 1000

```

```

C      COMPUTE THE LAGUERRE STEP CDIR AND THE BOUND FEJER AT ZN...
CALL CQDRTC( CMPLX( X2N1*ACF2(1) , X2N1*ACF2(2) ) ,
$          CMPLX( X2N1*ACF1(1) , X2N1*ACF1(2) ) , CF, CDIR, CF1 )
FEJER = ABS(ACDIR(1)) + ABS(ACDIR(2))
C = CMPLX( XN2N*AC(1) , XN2N*AC(2) )
C = C*CDIR
C = CMPLX( AC(1) + XN1 , AC(2) )
CDIR = CDIR/C
ABDIR = ABS(ACDIR(1)) + ABS(ACDIR(2))
FEJER = AMIN1( RTN*ABDIR , FEJER )
C      IS THE STEP SIZE NEGLIGIBLE . (THIS TEST MAY BE REDUNDANT )
DX = DABS(DZNR) + DABS(DZNI)
IF( DX + ABDIR .EQ. DX ) GO TO 1002
C      NOW DETERMINE WHETHER CDIR IS ACCEPTABLE .
GO TO 701
C
950 CONTINUE
C      FACTORIZATION OF POLYNOMIAL BY LINEAR FACTOR (Z-X) AS FOLLOWS
C
C          SUM(DU(I)*Z**(N-I)) = (Z-X)*SUM(Z(I)*Z**(N-I-1)) + Z(N)
C          FOR ALL Z ,
C
C      SO Z(N) IS VALUE OF POLYNOMIAL AT Z=X ,
C      FIRST DERIVATIVE OF POLYNOMIAL AT Z=X IS V , AND
C      SECOND DERIVATIVE OF POLYNOMIAL AT Z=X IS 2*W .
C      E IS ERROR BOUND FOR THE VALUE OF POLYNOMIAL AND
C      Z(I) ARE THE COEFFICIENTS OF QUOTIENT POLYNOMIAL .
OVFLOW = .FALSE.
C      BE SURE THAT THE OVERFLOW INDICATOR IS TURNED OFF.
DX = DZNR
DZNI = 0.00
ABX = ABS(X)
DV = DU0
W = 0.
C      CALL FPTUN TO REDUCE ERRORS CAUSED BY INTERMEDIATE UNDERFLOWS .
CALL FPTUN
Z(1) = DU(1) + DX*DU0
E = ABS(Z(1)) + ABX*ABS(DU0)
DO 951 I = 2,N
    W = V + X*W
    DV = Z(I-1) + DX*DV
951  Z(I) = DU(I) + DX*Z(I-1)
    FN = ABS(Z(N))
F = SNGL(Z(N))
IF(OVFLOW) GO TO 102
E = ABS(DU0)
DO 952 I = 1,N
952  E = ABS(Z(I)) + ABX*E
E = UN10*E
IF(OVFLOW) E = XN*E

```

Note: CQDRTC(CA,CB,CC,ZS,ZL) solves  $CAZ^2 + CBZ + CC = 0$  for  
 ZS and ZL given complex coefficients CA,CB,CC.

```

C
C CHECK WHETHER AN ACCEPTABLE ZERO HAS BEEN FOUND .
IF( FN .LE. E ) GO TO 1051
C IF FN HAS UNDERFLOWED, GIVE THE MESSAGE UVFUNF .
IF( AND( BIT , FN ) .NE. 0. ) GO TO 953
CALL UNCL(73, UVFUNF )
GO TO 1050
953 CALL FPTST
C HAS THE FUNCTION VALUE DECREASED .
IF( FN .GE. FO .AND. STARTD ) GO TO 1100
C
C OVFLOW = .FALSE.
C FIND THE LAGUERRE STEP AT DZNR .
R = V/F
C IF OVFLOW, A ROOT OF POLY. IS WITHIN 4*N*( SMALLEST NO. ) OF ZN.
IF(OVFLOW) GO TO 1050
CALL QDRTC( X2N1*W , X2N*V , F , C , CF1 )
C CALCULATE THE FEJER BOUND FOR SMALLEST ZERO .
FEJER = ABS(AC(1)) + ABS(AC(2))
R = XN2N*R
CDIR = C/CMPLX( R*AC(1) + XN1 , R*AC(2) )
ABDIR = ABS(ACDIR(1)) + ABS(ACDIR(2))
FEJER = AMIN1( RTN*ABDIR , FEJER )
C IS THE STEP SIZE NEGLIGIBLE .
DX = DABS(DZNR)
IF( DX + ABDIR .EQ. DX ) GO TO 1052
C NOW DETERMINE WHETHER CDIR IS ACCEPTABLE .
GO TO 701
C
C ACCEPT CZN AS A COMPLEX ZERO .
1000 CONTINUE
C SET UNDERFLOW INDICATOR TO .TRUE. WHEN FN UNDERFLOWS
UNF = .TRUE.
C PUT COEFFICIENTS OF QUOTIENT POLYNOMIAL IN DU ARRAY .
C ENTRY POINT WHEN FN HAS NOT UNDERFLOWED .
1001 CALL FPTST
C ENTRY POINT WHEN STEP SIZE IS NEGLIGIBLE .
1002 DU 1003 I = 3,N
1003 DU(I-2) = Z(I-2)
C DUO IS UNCHANGED FOR THE DEFLATED POLYNOMIAL.
Z(N) = DSIC( CMPLX( RND(DZNR) , RND(DZNI) ) )
Z(N-1) = DSIC( CONJG( Z(N) ) )
GO TO 201
C
C ACCEPT ZN AS A REAL ZERO .
1050 CONTINUE
C SET UNDERFLOW INDICATOR TO .TRUE. WHEN FN UNDERFLOWS
UNF = .TRUE.
C PUT COEFFICIENTS OF QUOTIENT POLYNOMIAL IN DU ARRAY .

```



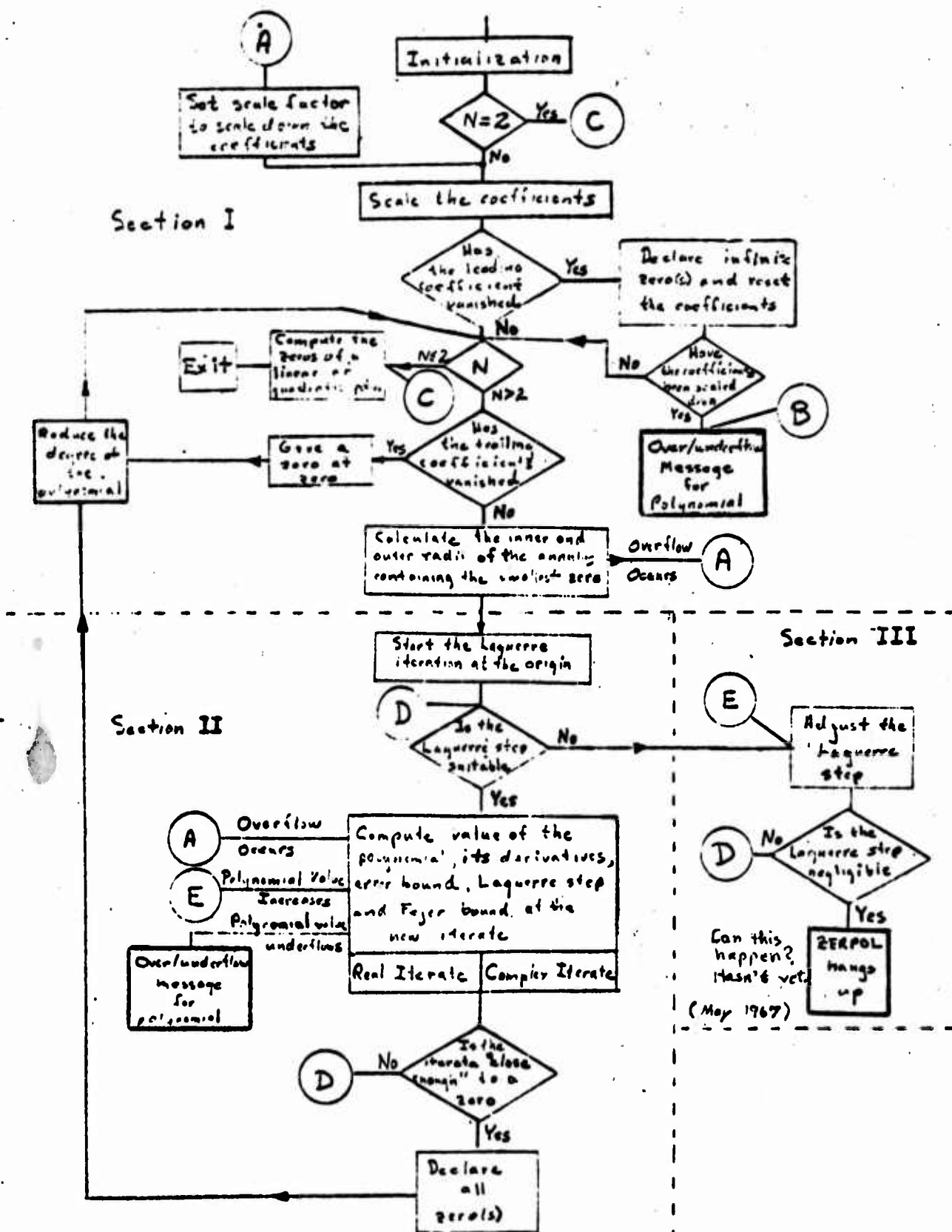
```

C      ENTRY POINT WHEN FN HAS NOT UNDERFLOWED .
1051 CALL FPTST
C      ENTRY POINT WHEN STEP SIZE IS NEGLIGIBLE .
1052 DO 1053 I = 2,N
1053   DU(I-1) = Z(I-1)
C      DUO IS UNCHANGED FOR THE DEFLATED POLYNOMIAL.
      Z(N) = RND(DZNR)
      GO TO 202

C
C      CURRENT LAGUERRE STEP IS NOT ACCEPTABLE .
1100 CONTINUE
C      IF STARTD, REDUCE PREVIOUS LAGUERRE STEP BY HALF.
      IF( .NOT. STARTD ) GO TO 1200
      ABSCL = 0.5*ABSCL
      CL = CMPLX( 0.5*ACL(1) , 0.5*ACL(2) )
C      HAS THE STEP BECOME NEGLIGIBLE .
      DX = DABS(DZNR) + DABS(DZNI)
      IF ( DX + ABSCL .NE. DX ) GO TO 801
C      OTHERWISE, ZERPOL HAS HUNG-UP.
      IF( FN .LT. E* $XN^{**2}$  ) GO TO 1103
      CMESH = CI12(N)
      CALL UNCLE( 75, MESH )
1103   IF(DZNI) 1002, 1052, 1002
C
1200 CONTINUE
C      IF .NOT. STARTD, HAS CZN BEEN ON THE INNER CAUCHY RADIUS.
      IF(SPIRAL) GO TO 1201
C      SET SPIRAL TO .TRUE.. PUT ZN ON THE INNER CIRCLE OF THE
C      ANNULUS CONTAINING THE SMALLEST ZERO IN THE DIRECTION OF THE
C      LAGUERRE STEP .
      SPIRAL = .TRUE.
      CSPIR = CMPLX( -1.25/ $XN$  , 1. )
      ABSCL = RU/ $XN^{**2}$ 
      C = CMPLX( (ACDIR(1)/ABDIR)*RU , (ACDIR(2)/ABDIR)*RU )
      GO TO 1202
C
C      SET ZN TO ANOTHER POINT ON THE SPIRAL .
1201 C = CSPIR*CMPLX( DZNR , DZNI )
1202 DZNR = AC(1)
      DZNI = AC(2)
      GO TO 802
      END

```

FLOW CHART 3.1



## 7094-II SYSTEM SUPPORT FOR NUMERICAL ANALYSIS\*

W. Kahan  
Department of Computer Science  
University of Toronto

**ABSTRACT.** This is the first half of a progress report on the author's efforts to improve the performance of IBSYS in the following areas of FORTRAN IV programming:

1. Error-traces and diagnostic messages to locate and explain flaws found while executing FORTRAN programs.
2. Post-mortem facilities via the FORTRAN IV statement  
IF (KICKED(OFF))... .
3. A consistent, sane and flexible treatment of over/underflow and related phenomena.
4. Digit manipulation (like rounding) via FORTRAN built-in functions.
5. The eradication of anomalies in the compiler (IBFTC) and the FORTRAN library (IBLIB).
6. The expansion of the FORTRAN library to include reliable and convenient subprograms for the solution of standard numerical problems like systems of linear equations,  
polynomial equations,  
eigenproblems,  
minimax approximation,  
fitting data by least squares,  
systems of ordinary differential equations,  
etc.

Items 1 to 5 are herein regarded as essential prerequisites to the accomplishment of item 6 in such a way that users of these subprograms need not supplement their own competency in mathematics, science, engineering or the humanities by a hyperfine proficiency at both numerical analysis and the debugging of systems programs. Each of the six areas will

---

\*This article previously appeared in SHARE SSD No. 159. We wish to thank the editors of SHARE for permission to publish it in these Proceedings.

be discussed in a correspondingly numbered section of this report, which begins by introducing the motivations for and the constraints upon the author's efforts. Sections 1 to 3 follow; section 4 to 6 will be issued separately later.

INTRODUCTION. For as long as electronic computers have been in use (since 1949 at the University of Toronto), there has existed a steadfast policy to widen the range of intellectual disciplines that might benefit from the machine. That policy is partly responsible for a decline in the numerical sophistication of users which has yet to be compensated by an increased sophistication in the programs they can use. Despite intensive attempts to educate them in the arts of computation, too many new users attribute to the numerical library subprograms the infallibility of a mathematical proof. They shall be disillusioned. To what extent can their disillusionment be written off as part of their education? To what extent can their dissatisfaction be traced to shoddy computing systems? There is room for improvement in both the quality of education and the quality of computer performance. But you cannot teach an old dog new tricks, and you cannot teach a new dog very much. Therefore the bulk of the improvement must and can come in the performance of computer systems.

The performance of IBM's IBSYS on the 7094-II has left a lot of room for improvement. The improvements listed here were motivated almost entirely by the inadequacies uncovered during the author's researches into numerical methods. The object of the researches was to produce working programs about which might be proved something simple and useful to a numerically unsophisticated but otherwise intelligent and educated user. As a by-product of these researches, the following vague generalities have emerged:

- Computation costs most when its result is not known to be right nor wrong, because it costs so much to find out what is wrong and why. Costs can be cut by a small amount of self-doubt applied early.
- Whether or not the purpose of computing be "insight", its most dependable benefit is hindsight. Programmers dislike forgoing this benefit through lack of foresight.
- Errors, anomalies and arbitrary restrictions hurt most when they are too rare to remember but not rare enough to ignore.

These generalities have influenced the many decisions on questions of detail which arose during the work on the system. A more decisive influence was exerted by three constraints:

First, it was deemed essential that programs be capable of conversion to whatever machine might replace the 7094-II, and so it was decided that all numerical subprograms be written in a language like FORTRAN or ALGOL, except where efficient coding was so obviously machine dependent that the assembly language MAP was used. I chose FORTRAN IV in preference to ALGOL. I would rather fight than switch. I am still fighting with the latest version (13) of the IBFTC compiler to incorporate all the modifications which I had introduced into the previous version, and further modifications to correct newly discovered deficiencies.

Second, since no one had anticipated a need to rewrite IBSYS or IBFTC in its entirety, no resources were allocated for such a task. Therefore, IBSYS and IBFTC have been modified as little as possible, instead of being replaced. The modifications have cost about three man-years of work all told, much of which has been dissipated in the transfer of the modifications from version 12 to version 13 of IBSYS.

Third, but most important, is our decision that the Toronto version of IBSYS remain compatible with the standard IBM IBSYS. Consequently, any FORTRAN IV program, even if it be in the form of a binary object-program deck, which has been designed for and runs correctly on a 7094 under standard IBM IBSYS with a hundred or so storage locations to spare runs at least as well under our modified system. If the program be recompiled with no other modification then the user may benefit from our improved diagnostics, especially where division by zero is concerned. Most of the users of our 7094-II are unaware of any departure from standard. But programs which run well on our system sometimes fail mysteriously at other 7094 installations.

In this report an attempt will be made to discriminate between IBM's standard IBSYS and our modified IBSYS by referring to theirs in the past tense whenever it differs from ours. Further details about IBM's IBSYS can be obtained from their manuals:

C28-6248	(IBSYS monitor)
C28-6389	(IBJOB; loader and library)
C28-6390	(IBFTC FORTRAN compiler) .

Further details about our modified system can be found in "The Programmers' Reference Manual" 2nd ed. obtainable from

The Secretary, Institute of Computer Science,  
University of Toronto,  
Toronto 5, Ontario,  
Canada.

and henceforth referred to as the PRM. Program listings are obtainable too if requested by name.

1. ERROR-TRACES AND DIAGNOSTIC MESSAGES. It may seem peculiar that a Numerical Analyst be preoccupied with the System Programmer's traditional responsibility for error-traces, diagnostics and post-mortem information. But let us watch the Numerical Analyst at work. Much of his computer time is dissipated by the diagnostics and post-mortems which he receives while trying to discover why his algorithms do not work as well as he had hoped. From time to time he hands one of his subprograms on to some other user numerically less sophisticated than himself, and in so doing he tacitly shares with the Systems Programmers some responsibility for issuing diagnostics. His program may produce diagnostic messages for different reasons than merely to signal its own collapse. Diagnostics may be the only "correct" answers that the program can deliver in response to problems outside the intended domain of its applicability, especially when the program's domain cannot easily be defined other than by attempting to execute the program. For example, a hopelessly ill conditioned linear system

$$A \underline{x} = \underline{b}$$

is most easily identified when a sound linear-equation-solver fails to solve the system for  $\underline{x}$  but exhibits instead a near linear dependence  $\underline{d}$  in the left hand side  $A$ ; i. e.

$$\| A \underline{d} \| / (\| A \| \| \underline{d} \|) \div 0 .$$

The Numerical Analyst's subprogram ought to pass on this kind of diagnostic information in a form easily interpreted either by the user's calling program or by the user personally.

The later form of diagnostic is usually a message printed amidst the user's output and is often the consequence of an error or oversight. The crucial question is

"Where was this error committed?"

but no computer program can answer this question. The best that can be done automatically is to answer the question

"Where did the program first encounter some anomalous consequence of the error?"

The answer takes the form of an Error-Trace. Under IBM's IJOB this would be provided by library subprogram .FXEM., the FORTRAN execution Error Monitor. Let us examine an error-trace typical of those produced by IBM's .FXEM.. For example, suppose line 2 of the user's main program MAIN called a subprogram SUB1 in whose line 25 was a call to SUB2 in whose line 17 was a reference to SQRT(-4.0). When this reference was executed, the SQRT program would detect the inappropriately negative argument and call .FXEM. (say in line 31) to produce an error-trace and diagnostic message. IBM's error-trace would look like this:

#### ERROR TRACE CALLS IN REVERSE ORDER

CALLING ROUTINE	IFN OR LINE NO	ABSOLUTE LOCATION
SQRT	31	17621
SUB2	17	14513
SUB1	25	07762
MAIN	2	05413

The names in the first column are the deck-names assigned by the user to his subprograms (or else, in our modified system, assigned by default by the system). The line numbers or "Internal Formula Numbers" in the second column refer to numbers printed in the programs' source listings, and can be exploited by the FORTRAN IV programmer without recourse to storage maps. For this reason, the third column of absolute octal core locations is of secondary value to the FORTRAN programmer. It is a great convenience that he can ignore this column and dispense with storage maps most of the time.

The completeness of the error-trace shown above is one of its most valuable features. Complicated programs can contain several references to the SQRT subroutine, and it is vital that the path of control to the invalid reference be laid out explicitly. The complete error-trace is even more valuable when languages which permit recursive procedures are used. If a user were instead provided with only the reference to SQRT (or only to SQRT and SUB2) in the error-trace above, he might waste a lot of time checking through all of his calls to SUB2 in an attempt to uncover the faulty one.

IBM's .FXEM. would print out a two-line diagnostic message and provide a means to exercise options regarding kick-off or continued execution following the diagnostic and error-trace. But .FXEM. suffered from two defects.



One, the easiest to remedy, was that `.FXEM.` could be called only from a MAP assembly language program. We fixed this by providing a program called UNCLE; any programmer can kick himself off (and produce an error-trace plus post-mortem debugging output) by executing

CALL UNCLE .

He can offer users of his program a limited range of kick-off-or-continue options by writing

CALL UNCLE (N)

with a suitably chosen integer expression N. He can supply one or two diagnostic messages too by writing

CALL UNCLE (N, Message)          or

CALL UNCLE (N, Message 1, Message 2) .

The messages can be inserted literally as Hollerith strings or they can be referenced as arrays of alphanumeric data. In the latter case, rudimentary binary-to-BCD conversion facilities are available to permit integer valued variables like indices or error-codes to be inserted into the diagnostic without first reserving core storage for the panoply of FORTRAN input/output subprograms. This last is an important consideration when program overlay is required during execution. (For more details about UNCLE, consult the PRM.)

`.FXEM's` second defect was that it could cope only with what I call "scheduled errors"; these are errors each of which is discovered in a subprogram which, when it calls `.FXEM.` to produce an error-trace, can supply whatever linking information is needed by `.FXEM.` to start the error-trace. For example `SQRT(-4.0)` is a scheduled error because `SQRT` is called in a standard way. But when unscheduled errors like over/underflow, division by zero, running overtime, ... , were detected they would "trap", i.e. cause interrupts which transferred control to appropriate subprograms without carrying the standard linking information that made an error-trace possible. Consequently, the diagnostics for unscheduled errors answered the question "where?" with an absolute octal core location, but could not answer the question

"How did I get there?"

That IBSYS's standard linking sequence contained a partial answer to the last question was widely recognized. The first effort to extract a full



answer was made by G. Wiederhold and G. D. Johnson at Berkeley (Univ. of California) in 1963. Their work has appeared in SHARE SSD 121 of May 21/64 and SDA's 3066-7. A similar scheme was devised by J. Leppik, G. Howard and the author at Toronto in 1964. Our scheme differs from theirs mainly in that ours is simpler to use, slightly less flexible, and fully compatible with the standard IBM system.

The first step in both schemes is to revise the standard SAVE pseudo-operation by which subprograms are expected to save and restore index registers, control linkages, etc. When IBM's SAVE was executed upon entry to a subprogram SUB, it used to save in a cell called SYSLOC the pointer to the statement

CALL SUB ,

but no subsequent use was made of SYSLOC. We have added two instructions to SAVE whose effect is to store the same pointer, during the RETURN from SUB to the instructions following

CALL SUB ,

in such a way that the contents of SYSLOC show whether SUB has just been entered or has just returned. This modification has no effect upon the way IBM's .FXEM. behaves for scheduled errors.

Next, I rewrote .FXEM. so that it can be called from a trap-handling program. Such a CALL is distinguished from other standard CALLS by the absence of certain otherwise expected linking information, the lack of which forces .FXEM. into a new mode of action which examines SYSLOC to produce the first line of the error-trace.

The behaviour of the new .FXEM. is best illustrated by an example. Suppose that SUB2 in the example above contains, besides SQRT(-4.0), a division which, when executed, turns out to be a division of zero by zero. The result is the following diagnostic (in which the contents of the second line depend upon an option selected by the user):

0.0/0.0 ERROR AT 14506

RESULTS IN 0.0 or EXECUTION TERMINATED

ERROR-TRACE WITH CALLS IN REVERSE ORDER CODE 25

CALL IS IN DECK NAMED	AT IFN OR LINE NO.	ABSOLUTE LOCATION
SUB2	17+	14513
SUB1	25	07762
MAIN	2	05413

The important change shows up in the + sign after the line no. 17. This means that the announced anomaly was detected during or after (in time) the execution of line no. 17 of SUB2, but before any subsequent CALL was executed. Since SUB2 has a call to SQRT in line 17 at location 14513 (cf. the previous error-trace), and the 0.0/0.0 occurred five words ahead of this location in the program, it seems likely that the program was executing a loop, perhaps a DO-loop, which contains the offending division just a line or two in the listing ahead of the square root; and this loop was executed at least once before the divisor vanished.

The detective work in the last sentence is not typical; usually the error can be located by the most superficial inspection. But the need for any detective work at all is an unfortunate consequence of the way IBM's FORTRAN IV compiler works. Instead of identifying every line in the symbolic listing with a line number that .FXEM. could deduce at execution time (for example, by locating a dummy instruction

TX ID, O, LKDR

at the beginning of the coding emitted by the compiler for line no. ID of the FORTRAN subprogram whose linkage information can be found at LKDR), the compiler assigns a useable line number only when a CALL is generated. Since an implicit CALL is generated for all references to FUNCTION subroutines, as well as for most exponentiations of the form  $X^{**}J$  and  $X^{**}Y$ , for input/output, for complex multiplication and division, and for a computed GO TO( $n_1, n_2, \dots, n_m$ ), I, there are few programs whose listed line numbers are too sparse for a successful interpretation of the error-trace. And, at worst, the unscheduled error is located to within one subprogram.

The CODE 25 at the head of the error-trace tells the programmer how to exercise his option to define 0.0/0.0 in one of two ways; either

0.0/0.0 = 0.0 and continue execution, or  
0.0/0.0 = EXECUTION TERMINATED.

For example, the first option is the result of executing

CALL KIKOPT (25, 1)

while the second results from

CALL KIKOPT (25, 0) .

The reader is referred to the PRM for precise details about available options and how to exercise them conveniently. What follows is a condensation.

The PRM contains a table of error codes and messages (cf. Fig. 25 and the section "Subroutine Library Error Messages" in IBM's IJOB manual, Form C28-6389-1) which describes for each code its error condition, the options available, and which option is assumed by the system in default of a request to the contrary. The default option is usually to provide a message and then continue execution in some reasonable way.

I believe that, taken together with the other diagnostic facilities in our system, our surprisingly simple set of options covers almost all circumstances satisfactorily. For serious errors we assign positive codes, like +25 for 0.0/0.0, to signify that the allowed options are

- +1) Give a message and error-trace, and then continue reasonably,  
or
- +0) Give a message and error-trace, and then terminate execution,

(Some errors, like

GO TO (1, 2, 3), 4

are so serious that option +1 is denied.) For milder errors we assign negative codes, like -13 for SQRT (-4.0), which signify that the allowed options are

- 1) Give a message and error-trace, and then continue reasonably,  
or
- 0) Give no message nor error-trace; just continue reasonably.

The meaning of "continue reasonably" is discussed later in this report. For now it suffices to give a few examples:

<u>Error Condition and "Reasonable" Response</u>	<u>Code</u>
$\text{SQRT}(-X) = -\text{SQRT}(X)$	-13
$\text{LOG}(-A) = \text{LOG}(\text{ABS}(A))$	-10
$0.0**0 = 1.0$	- 3
$0**0 = 1.0$	- 1
$0.0**0.0 = 1.0$	+ 6
$0.0/0.0 = 0.0$	+25

\*Footnote: We allow programmers to write  $\text{LOG}(X)$  or  $\text{ALOG}(X)$  interchangeably as they please rather than penalize them for the venial sin of omitting the  $A$ .

Programmers, particularly writers of library subprograms, can easily provide other kinds of optional responses to error conditions detected by their own subprograms because the status of the option-indicator (a binary digit) associated with any error-code number can be sensed and stored as well as change via  $\text{KIKOPT}$ . A complicated program may have several error-codes assigned to it, but this causes no problems because 280 codes are available. Programmers are free to use error-codes as flags or flip-flops in a way comparable to the use of sense-switches and sense-lights on the older slower machines.

A comment is required to explain that last  $\text{.FXEM.}$  option -0 which, in effect, allows  $\text{.FXEM.}$ 's activity to be suppressed entirely when the error is a mild one with a negative code. Some of these errors are better described as differences of opinion about the most apt definition of a function or an expression, as in the cases of  $0**0 = 1$  and  $0.0**0 = 1.0$  (cf. the Taylor series  $\sum_{r=0}^{\infty} a_r x^r$  at  $x=0.0$ ). In these cases the warning messages serve only to remind users that my definitions are not universally accepted in the computing world. If he is satisfied to do things my way, he can turn the message off. If he prefers another way, he can easily change the relevant program to his own specifications with the aid of the documentation which we supply.

Other errors with negative codes sometimes represent minor oversights; an example is

$$\text{LOG}(-X) = \text{LOG}(\text{ABS}(X)) \quad , \text{ Code } - 10.$$

For reasons discussed later, our policy is to try not to terminate execution because of such an oversight. Rather, it seems better to continue and find out what else the programmer overlooked. We do not encourage programmers to exploit system side-effects to save the bother of a sign-test or

some such simple instruction. We do not regard the -0 option as one which should be employed in production or library programs to correct oversights, except possibly temporarily, because this type of hidden coding is so difficult to remember when late-hatching bugs are being sought.

To implement the new .FXEM. and error-trace required several man-months of work, most of which was spent tracking down anomalies. For example, several input/output programs supplied as part of earlier versions of FORTRAN IV were found to use non-standard subprogram linkages, and these had to be repaired to allow even the old .FXEM. to produce meaningful error-traces before they were further modified to work with the new .FXEM. . Every library program had to be examined; here we reaped an unexpected reward when we discovered that the new .FXEM. makes possible a shorter and faster subprogram linkage to certain library programs like SQRT, SIN, COS, LOG, EXP, complex multiply, complex divide, A\*\*J, and others.

But one large job remains. The FORTRAN compiler must be modified to generate standard CALLs to Arithmetic Statement Functions which at the present, as compiled by IBM's FORTRAN IV v. 13, use non-standard CALLs in order to save about 7 microseconds per CALL. (One division costs 8.4 microseconds.) Consequently both IBM's .FXEM. and ours produce error-traces which skip, sometimes confusingly, over references to Arithmetic Statement Functions.

2. POST-MORTEM FACILITIES. We prefer to think of kick-off as an act of desperation on the part of a subprogram, and therefore try not to terminate execution unless it is overwhelmingly probable that continued execution will be an utter waste. There is little risk that errors like SQRT(-4.0) will be repeated millions of times to no good purpose, because the monitor imposes the user's own limit upon the total number of lines of printed output, thereby protecting him from a million lines of SQRT's diagnostic and error-trace. Furthermore, programmers who are especially sensitive to a waste of their computer time allotment can use statements like

```
IF (CLOCK (TSTART) .GT. TMAX) CALL UNCLE
```

to kick themselves off when the elapsed time since

```
TSTART = CLOCK (0.0)
```

exceeds TMAX, at a cost of 70 microseconds per execution. (One square root costs 64 microseconds.)

But sometimes kick-off is the only reasonable response to an error. This response gives rise to a class of programmer who has only one diagnostic and error-trace to show for his several seconds (or minutes) of computer time. It is uncharitable to advise him that he should have exercised enough foresight to provide intermediate output as insurance against such an event. Besides, he may reply

"I thought I had debugged that program!"

We doubt the wisdom of the widespread tendency to inundate every user who is kicked off with a complete dump of storage willy-nilly. This could drown him in octal data which he is unlikely to be able to read. It is a costly way to educate students.

The ideal solution would be to display conveniently just those variables which have figured in the events leading up to the debacle. Our solution is not ideal, but it is simply and flexible. It is an improved version of our PMORT described in Comm. A.C.M. 7 (1964) p. 15. We allow the programmer to write into his FORTRAN IV program a statement of the form

```
IF (KICKED(OFF))    <any executable statement>
                   < the next executable statement >
```

with the expectation that, because the value of the logical function KICKED is always .TRUE., his program will merely execute <the next executable statement> . But if and when his program is kicked off, the monitor will give him the diagnostic and error-trace that he deserves and then, after over-writing <the next executable statement> with CALL EXIT, will execute <any executable statement> .

e.g. 1:        IF(KICKED(OFF)) WRITE(...)

causes the desired information to be written out if and only after the program has been kicked off. The programmer can choose a FORMAT to suit himself or, if more convenient, he can use the simple unformatted output provided by the NAMELIST feature of FORTRAN IV; or he can CALL DUMP and be drowned.

e.g. 2:        IF(KICKED(OFF))    CALL ...        or  
                                  GO TO ...

causes the desired transfer of control to take place after kick-off, and thus permits a user to store valuable data on magnetic tapes and ask the operator to save them. Or he can call a complicated diagnostic program of his own, or he can try again to solve his problem by some method other

than the one which failed. The monitor will allow, say, 20 seconds and 300 printed lines of computer activity after the first kick off. Of course, any second kick-off is final despite further IF (KICKED(OFF))... requests. Because the user has recourse to KICKED, writers of library and systems programs are under less pressure when they have to decide whether an anomalous condition should terminate execution or just produce a warning.

Programmers are encouraged to use KICKED as often as they like in both FORTRAN and MAP assembly language programs, and they can leave these KICKED statements in production programs as insurance against the remote possibility that an undiscovered bug may terminate execution in a cloud of mystery. Each executed reference to KICKED consumes less than 14 microseconds (less than two division times) so KICKED can be used in fairly tight loops without seriously wasting time. The monitor will respond at kick-off only to the last executed reference to KICKED.

An important limitation upon KICKED was imposed by the absence of any block structure in FORTRAN comparable to that in ALGOL, and by the way that indexing is optimized in FORTRAN. This limitation exists because, whenever kick-off occurs in some subprogram remote from the one containing the KICKED statement and then control is passed to <any executable statement> after the IF(KICKED(OFF)), no attempt is made to restore index registers to the state they were in when KICKED was called nor to re-set tapes to their former positions. More important, there is no way to reproduce the effect of those instructions which may have been placed in "optimum" positions ahead of the call to KICKED in order to initialize index registers and addresses as efficiently as possible from the point of view of the normal sequence of control. For example, if kick-off occurs during the computation of FCN in the sequence

```

DO      3      J = 1, 10
  A(1, J)      = J - 1
DO      3      I = 1, J
  IF (KICKED(OFF)) WRITE(...) I, J, B(I), B(J), (A(K, J), K=1, J)
3      A(I + 1, J) = FCN(B(I), B(J), A(I + 1, J)) + A(I, J)

```

there is no way at kick-off time to move the numbers I and J from storage into the appropriate cells and index registers for the references to B(I), B(J), A(K, J) and "K = 1, J" following the call to KICKED.

A second limitation shows up when program overlay takes place; there is no simple way to detect whether <any executable statement> in the IF (KICKED(OFF)) statement has been partially overlaid, or whether it refers to data which has been overlaid. Consequently we inserted an instruction in .LOVRY, the overlay handling subprogram, which causes the

Whenever the array Y is changed, indicate which element too;

Y (2) = .74131042 E - 18 .

Whenever the third column of array Z is changed, say so;

Z(13, 3) = 0.0 .

Whenever the subprogram PROG is called, write out its arguments;

CALL     PROG (13, 27.421493, Y) WITH

Y(1) = 1.4012362

Y(2) = .74131042 E -18

Y(3) = 0.0 .

IF PROG is a function, write out its value too;

PROG (13, 27.421493, Y) = 1.7014 E38 WITH

Y(1) = etc.

Whenever statement n is executed, say so. If this is a logical IF statement, tell what happened.

The MONITOR facility as described above has been implemented at least partially in several compilers; unfortunately, ours is not one of them. The problem is to deal with the statement

IF (KICKED(OFF)) MONITOR ..... ,

for which the nicest solution would be a retroactive display of, say, the last 300 lines of output which would have been produced if that MONITOR statement had not been bypassed. Some compilers already have a feature of this kind; the author envies their users.

Now is a good time to compare the error-options needed by the programmer with those available to him. He may want to assign to a specified anomaly, like 0.0\*\*0 , one of the following four consequences:

- 0) Re-interpret the request in a way judged to be appropriate for the majority of users (say 0.0\*\*0 = 1.0) and continue with no message nor error-trace.
- 1) Re-interpret the request as above, and put out a message and error-trace to tell the programmer what happened and where, and then continue execution.



- +0) Put out a message and error-trace to explain where and why execution was terminated, and then grant any post-mortem request that may have been made via

IF (KICKED(OFF)) ...

- 2) Transfer control to a location designated in advance by the programmer where he may cope with the anomaly as he pleases, provided the necessary information is easily accessible to him.

Our system offers at least two of the first three options for most error conditions. The last option is dangerous in FORTRAN for the reasons cited while discussing the limitations of KICKED, unless it is handled carefully. The following discussion explains how some of our library programs offer option 2).

Consider for example our least squares library subroutine LSTSQ which, given a rectangular  $M \times N$  matrix  $X$  and a column vector  $y$ , attempts to find that coefficient vector  $c$  which minimizes the sum of squares

$$S = (y - Xc)^T (y - Xc) = \sum_i (y_i - \sum_j x_{ij} c_j)^2.$$

A solution  $c$  always exists and satisfies the normal equations

$$X^T X c = X^T y.$$

LSTSQ tries to solve these equations (in double precision, because that is the fastest adequate method on a 7094) for  $c$  and the corresponding minimum value of  $S$  and, if requested, the inverse matrix

$$V = (X^T X)^{-1}.$$

But if the columns of  $X$  are nearly linearly dependent, in the sense that there exists a perturbation  $\Delta X$  of the order of a few units in the last place of  $X$  such that the columns of  $(X + \Delta X)$  are linearly dependent, then the solution  $c$  is not well defined and LSTSQ produces one of two things instead of  $c$ .

- 0) If the user wrote

CALL LSTSQ (X, M, N, Y, C, S)    or  
CALL LSTSQ (X, M, N, Y, C, S, V)

then he has made no provision for the possibility that  $X$  be nearly singular, so he receives a suitable diagnostic and error-trace and is kicked off.

1) If the user wrote

```
CALL LSTSQ (X, M, N, Y, C, S, $n)    or  
CALL LSTSQ (X, M, N, Y, C, S, V, $n)
```

where  $n$  is an integer standing for a statement number, LSTSQ returns control to statement number  $n$  in the user's calling program, and diagnostic information is made available in  $V$  (or elsewhere if  $V$  was not requested) which permits the calling program to identify the linear dependence relatively easily and change  $X$  appropriately. (Usually the calling program just decreases  $N$ .) LSTSQ does not put out any messages in this case.

The foregoing description is somewhat simplified; details can be found in the PRM. The interesting feature is not so much the use of a FORTRAN IV error return  $\$n$  as the fact that this error return is optional. The option is available because one of the first statements executed within LSTSQ is

```
CALL ARGCNT (I, J)
```

which counts the arguments supplied in the CALL to LSTSQ.  $I$  is the number of arguments exclusive of error returns, and  $J$  is the number of error returns. The error options described above are numbered 0 and 1 according to the value of  $J$ . Similarly, LSTSQ determines whether the user wants  $V = (X^T X)^{-1}$  or not according as  $I = 7$  or 6 respectively. Any other values of  $I$  or  $J$  indicate an error, like a period between the integers  $M$  and  $N$  instead of a comma, which is serious enough to terminate execution with an appropriate diagnostic.

The use of variable length argument lists lends a certain elegant simplicity to several of our library programs, and we hope that this feature will be incorporated in the programming languages of the future. The simplicity with which the error return scheme can be implemented makes it efficient and satisfactory for a wide range of applications, but there are two important areas where the scheme is unsatisfactory. One consists of those difficulties caused by a small lack of foresight and recognized immediately with the slight assistance to hindsight provided by a diagnostic. Many of the error conditions mentioned above, like LOG( $X$ ) when LOG( $ABS(X)$ ) was intended, fall into this category. So do

many input/output problems. It suffices here to say that a lot more could be said for the desirability and convenience of subprograms like KIKOPT which allow the programmer to revise temporarily the execution of his program at each of several spots without having to insert a small explicit change at each spot.

The second area where error returns have proved unsatisfactory covers Over/Underflow, a ubiquitous phenomenon to which the next section of this report is devoted.

3. OVER/UNDERFLOW. Overflow and Underflow are what take place in the arithmetic registers of a computer whenever an attempt is made to calculate numbers outside the normal range. On the 7094, overflow occurs whenever the magnitude of the result of a floating point arithmetic operation equals or exceeds

$$2^{127} \doteq 1.70141183 \times 10^{38} ;$$

underflow occurs whenever the magnitude is not exactly zero and is smaller than

$$\bar{2}^{129} \doteq .146936794 \times 10^{-38} .$$

Special provision must be made to cope with over/underflow in a way which does not produce misleading results.

It is sometimes argued that overflow is an error for which the penalty should be

#### EXECUTION TERMINATED

but this penalty would place an intolerable burden upon even the most expert numerical analyst. He is often unable to predict in advance what the range of numbers will be in complicated calculations, especially where exponentials, polynomials and rational functions of high degree, or spaces of high dimensionality are concerned. For example, if  $P(x, y)$  is a polynomial in  $x$  of degree 10 whose coefficients are wild functions of  $y$ , then the desired solution  $x = X(y)$  of the equation  $P(x, y) = 0$  may be well-defined and reasonable even though it is inaccessible unless the polynomial-zero-finding subprogram is allowed to pursue a flexible scaling strategy in response to over/underflows, if any, which occur during the computation of  $P(x, y)$ . Overflows should not force kick-off; if worst comes to worst, a program can kick itself off by executing, say,

IF(OVFLOW) CALL UNCLE(0, 22H INESCAPABLE OVERFLOW.).

An opposite attitude of laissez-faire is reflected in the designs of those machines whose hardware automatically replace an overflowed magnitude by a special digit pattern representing  $\infty$  and then plunge on. Such a scheme might well include, say,  $\theta$  to replace an underflowed magnitude and  $\oplus$  to indicate an indeterminate value. These symbols might obey rules like the following:

- i) Whenever an arithmetic operation generates  $\pm\infty$ ,  $\theta$  or  $\oplus$ , a corresponding flag is raised to indicate to the program that overflow, underflow or lost significance respectively has occurred. If requested by the programmer in advance, a message can be printed out for his information.
- ii) Any arithmetic operation with  $\oplus$  as an operand generates  $\oplus$  as a result.  $\oplus$  is also generated by the following expressions:  $\infty - \infty$ ,  $\infty/\infty$ ,  $0/0$ ,  $0/\theta$ ,  $\theta/0$ ,  $\theta/\theta$ ,  $\infty * 0$ ,  $\infty * \theta$  and  $x/\theta$ .
- iii) If  $x \geq$  (1 unit in the last place of the overflow threshold)  
then  $\infty - x = \oplus$ ; otherwise  $\infty \pm x = \infty$ .  
If (1 unit in the last place of  $x$ )  $\leq$  (the underflow threshold)  
then  $\infty - x = \oplus$ ; otherwise  $x \pm \theta = x \pm 0 = x$ .  
If  $x \geq 1$  then  $x * \infty = \infty * \text{sign}(x)$ ; otherwise  $x * \infty = \oplus$ .  
Similar rules hold for  $x/\infty$ ,  $\infty/x$ ,  $x * \theta$  and  $\theta/x$ .  
 $x/0 = \infty * \text{sign}(x)$  unless  $x = 0$  or  $\theta$ .
- iv) The number 0 can be generated only by direct assignment or as the result of  $x-x$  with  $x \neq \theta$  nor  $\infty$ . The symbol  $\theta$ , which stands for the set of all numbers smaller in magnitude than the underflow threshold, can be generated only by direct assignment or by an underflow as indicated above. During comparisons the symbol  $\theta$  simultaneously satisfies

$$\theta \neq 0, \theta \neq \theta, \theta \neq 0, \text{ and}$$

$x > \theta$  if and only if  $x > 0$  too.

Rules like the foregoing are formidable, and have not been implemented in any hardware known to the author (who would not expect to find them in any machine except possibly one with interval-arithmetic built into the hardware). But no other less elaborate rules are known to be foolproof. For example, the CDC 6600's hardware follows similar rules whose most obvious difference is the lack of any distinction whatever between underflow to  $\theta$  and the number 0. A comparable deficiency is to be found at

those IBM installations where, to escape a plethora of insignificant underflow messages, all underflow messages are suppressed by many users most of the time. The following segment of FORTRAN coding shows what can happen when this is done. Here A, B, C, D and X are all positive normalized floating point numbers (not special symbols nor zero).

```

Y = (A*X+B)/(C*X+D)
Z = (A+B/X)/(C+D/X)
W = Y/Z
WRITE (...) W
.....
Output:  W = 1.98

```

In the absence of any indications of over/underflow, how is this phenomenon to be explained? The only thing unnatural about this example is the WRITE statement; W is more likely to have remained "out of sight, out of mind".

The replacement of underflowed numbers by zero with no indication to program nor programmer is a clearly unsatisfactory practice. And even when an indication of over/underflow is given, there is ample reason to protest against the destruction by hardware (as on the IBM 360 and CDC 6600) rather than software of information which could otherwise be of significance to the programmer; this is discussed in more detail below in connection with the Unnormalized Mode and the Counting Mode of treating over/underflow. But, to be fair, it must be acknowledged that most programmers would be satisfied most of the time by the provision of representations for  $+\infty$ ,  $-\infty$ ,  $\theta$  and  $\emptyset$  obeying rules like i) to iv) above.

What more might a numerical analyst demand? From time to time he will want to generate and use numbers which lie beyond the over/underflow thresholds. And certainly no programmer wants to be forced to check for over/underflow after (much less before) the execution of each arithmetic instruction in his program, and to decide each time upon an appropriate course of action. He will prefer to choose one of the several modes of execution provided for him by the system, with the understanding that while the program is being executed in his chosen mode each over/underflow will be treated according to the rules tabulated for that mode. Rules i) to iv) above could define one such mode. The programmer should be allowed to change modes between one line of his program and the next. Ideally, he should be allowed, if he wants, to define his own mode by specifying in detail just what rules are to be obeyed for each type of arithmetic operation. Finally, although the programmer who is ignorant of the problems of over/underflow must be warned when they occur, care must be taken not to drown him in a cascade of over/underflow messages, especially when they are obviously irrelevant. (An example of an obviously irrelevant underflow is remainder underflow after a floating point division in a

FORTRAN program, which always discards the remainder.)

An attempt has been made to serve as many of these needs as can be served in a FORTRAN context by means of a substantial extension of the service supplied by IBM via their subprogram .FPTRP in IBJOB. This program exploits the fact that whenever a floating point over/underflow occurs the 7094 "traps"; it interrupts itself and transfers control to a designated core location after setting up an indicator word (cell 0) to describe what caused the trap and where. This floating point trap, FPT, takes precedence over all others in the machine; and when it occurs the registers in the machine contain the over/underflowed result unaltered, so that no significant information is lost. A hardware option can be purchased (RPQ 880291) which includes improper divisions like  $1/0$  in the scope of the FPT.

I rewrote .FPTRP in a way which, while maintaining compatibility, increased its speed and augmented its capabilities so that programs can easily choose and change to any one of five modes of execution. The Standard Modes treat over/underflow very much as IBM did, the main difference being that now underflow sets up an indicator the same way as does overflow. The Unnormalized Modes exploit unnormalized arithmetic to permit underflow to occur "gently" without setting up distracting indicators or messages. The Silent Modes set indicators to indicate over/underflow to the program but put out almost no messages for the programmer; cascades of over/underflows in the Silent Modes do not slow programs down appreciably. The Printing Modes set indicators for the program and also report each indicated over/underflow, as it occurs, in a printed message for the programmer, thus helping him to debug his program. The Counting Mode allows certain kinds of computations to be carried out with no risk of over/underflow because the allowed range of magnitudes is extended to include numbers like

$$2(\pm 2^{42})$$

These five modes are discussed below in appropriately titled subsections of this report. The last two subsections discuss improper divisions and simulated over/underflows.

**THE STANDARD SILENT MODE.** This is the mode in which the system operates by default in the absence of requests for some other mode. Whenever a floating point arithmetic operation overflows, its result is replaced by the largest possible magnitude ( $1.7014 \times 10^{38}$ ) with the same sign, and this event is recorded by setting `OVFLOW = .TRUE.` . Whenever a result underflows it is replaced by zero with the same sign, and this event is recorded by setting `UNFLOW = .TRUE.` . The indicators

OVFLOW and UNFLOW are logical variables which can easily be sensed, stored and/or reset to .FALSE. in several ways described in the PRM. In particular, the declarations

```
LOGICAL OVFLOW  
COMMON /OVFLOW/OVFLOW
```

permit statements like

```
IF (OVFLOW) .... and  
OVFLOW = .FALSE.
```

to be executed without wasting time on subprogram linkages in short loops.

This mode is called Silent because each over/underflow sets its indicator without disturbing the programmer's output with any diagnostic message. However, just after his program's execution is terminated (either normally or by kick-off) a message is produced to draw the programmer's attention to any over/underflows that escaped the attention of his program; more about this later. In the Standard Silent Mode, each over/underflow costs 15 to 30 microseconds; i. e. two to four division times.

THE STANDARD PRINTING MODE. This mode differs from the previous mode only in that each over/underflow, as it occurs, inserts a message into the programmer's output to answer the following questions:

What happened, overflow or underflow?

Which machine registers are involved; AC, MQ or both?

What arithmetic operation was attempted; + , - , \* , / , double-precision, ..., ? (An octal operation-code is given here.)

What change was made in the affected register(s)?

Where is the instruction whose execution caused this over/underflow? (An octal core address is given here.)

Where in the source-program did all this happen?  
(An error-trace is given here by our version of .FXEM. .)

We also considered writing out the operands whose sum, product or quotient had over/underflowed, but the cost of doing so seemed more than



the information was worth. This point deserves reconsideration. Anyway, the error-trace usually points to within a few lines of the site of the over/underflow in a FORTRAN program.

The over/underflow handling subprogram .FPTRP can be switched in 40 microseconds from a Silent Mode to the corresponding Printing Mode via the statement

CALL NFPTST(M)

with a positive integer expression M. When this statement is executed, an internal counter N is set to M and .FPTRP is caused to operate in a Printing Mode until M over/underflow messages have been put out. N is decreased by 1 each time a message is put out, and when N becomes 0 an extra message

NOW OVER/UNDERFLOW MESSAGES ARE IN ABEYANCE

is produced and the Mode is switched back to Silent.

CALL NFPTST(0)

switches the Mode back to Silent without any extra message.

In accordance with current good practice, the FORTRAN programmer is allowed easily to sense, save, set and/or reset the message-counter N as well as the indicators OVFLOW and UNFLOW. Details may be found in the PRM. But programmers are advised not to set the latter two logical variables to .TRUE. directly in a FORTRAN program; instead they are advised to force an over/underflow like

DUMMY = (1.7E38)\*\*2 .

This is done because, whenever over/underflow occurs, .FPTRP stores the current contents of SYSLOC into the appropriate indicator to make it .TRUE. . Later, when the program's execution is finished, the monitor looks at each indicator to see whether it is .TRUE. , and if so then that indicator is interpreted as a pointer in roughly the same fashion as .FXEM. interprets SYSLOC when providing the first line of the error-trace immediately after an over/underflow in the Printing Mode. Consequently, the programmer's output finishes, whenever appropriate and possible, with a message like

LAST UNREQUITED OVERFLOW WAS IN OR AFTER  
LINE 17 OF DECK SUB2 .



LAST UNREQUITED UNDERFLOW WAS IN A SUBPROGRAM  
CALLED IN LINE 24 OF DECK SUB1.

Often the programmer can deduce from the information given here that the over/underflows did no harm; then, since the messages have not tainted his formatted output, he is free to cut them off and publish the rest.

If program overlay has intervened between the last unnoticed over/underflow and program termination, or if the indicators `OVFLOW` and `UNFLOW` were set to `.TRUE.` in a naive way, then the post-execution message may describe the desired deck-name and line number as `UNKNOWN`.

It is especially important to understand that the word "UNREQUITED" means that the program did not respond to the over/underflows and then reset the indicators to `.FALSE.` . The programmer may also have received several printed messages to notify him of each over/underflow that it ignored.

I see now that we could have supplied, at little extra cost, post-execution warnings more like this:

3943 OVERFLOWS WENT UNREQUITED BY THE PROGRAM  
BETWEEN LINE 17 OF DECK SUB2

AND A SUBPROGRAM CALLED IN LINE 64 OF DECK SUB1.

Such a message can be more useful in deciding whether or not to ignore the over/underflows. Also, the counts of overflows and underflows could be used by any programmer who, for reasons unclear to me, wished to terminate his program's execution after a specified number of overflows had occurred. Another improvement would be to allow a negative value for `M` in

`CALL NFPTST(M)`

to signify that `-M` overflow messages are to be allowed while all underflow messages are to be suppressed. Most of these improvements have been incorporated into the adaptation of our scheme for the Burroughs B5500 written by Mr. Michael D. Green at Stanford University in 1966, and I expect to put them into our system soon.

THE TREATMENT OF UNDERFLOW. Some programmers have good reasons to want to be informed about underflow. They may want to avoid consequent loss of precision or subsequent division by zero. But most

programmers whom I asked said they preferred that underflowed numbers be replaced by zero without their attention being distracted by the event. This attitude was justified at a time when most over/underflow messages reported "MQ UNDERFLOW" during an addition, subtraction, multiplication or double-precision division. This message signified that the double-length result of those operations in the AC-MQ register was small enough to cause the characteristic of the less significant word in the MQ to underflow even though the more significant word was correct. Since the less significant word is entirely ignored in single-precision FORTRAN expressions, and since the double-precision hardware of the 7094 ignores the characteristic of the less significant word in double-precision expressions, I decided that .FPTRP should simply ignore MQ underflow after those operations where it was obviously irrelevant.\* This decision's first consequence was a welcome reduction in the number of messages and complaints, especially where iterative calculations with residuals tending to zero were concerned. The second consequence was that certain old 7090 programs, which had performed double-precision arithmetic by simulating the 7094's double-precision hardware, ran into spurious overflow troubles and required revision so that they would use instead of simulate our machine's hardware. Fortunately, any user who insists upon running a 7090 program unchanged upon our 7094 can do so in safety by merely changing two well-marked instructions in .FPTRP. The second instruction is needed to force appropriate action when remainders underflow after division; otherwise they would be ignored too.

It is not good enough that the system ignores obviously irrelevant underflows. Many irrelevant underflows are not obviously irrelevant. Consider, for example, a segment of a typical matrix handling program which computes

$$r = b - \sum_i a_i x_i$$

The usual rule, which replaces each underflowed sum or product by zero, is satisfactory except when  $b$  and all the products  $a_i x_i$  are so close to the underflow threshold that the usual rule produces a significantly wrong value for  $r$ . If all underflows are reported, how can the rare significant reports be distinguished from the common ignorable ones? If no underflows are reported, how can the rare incorrect values of  $r$  be distinguished from the common correct ones? The easiest way I know to cope with these questions is to use our Unnormalized Modes:

---

\*The 27 significant bits in the MQ are not ignored nor cleared when the characteristic of the MQ underflows, so no accuracy is lost.

THE UNNORMALIZED SILENT MODE AND THE UNNORMALIZED PRINTING MODE. These two modes differ from one another in just one respect; the Printing Mode reports overflows in the way described under the Standard Printing Mode above. The two Unnormalized Modes differ from their corresponding Standard Modes only in the way they treat underflow. A number, which in a Standard Mode would have underflowed to zero and set UNFLOW = .TRUE., is in an Unnormalized Mode replaced by its closest unnormalized approximation and UNFLOW is unchanged. For example, consider a decimal machine whose underflow threshold is  $.10000000 \times 10^{-38}$ . In a Standard Mode,  $.15743219 \times 10^{-40}$  would underflow to zero, but in an Unnormalized Mode it is replaced by  $.00157432 \times 10^{-38}$ . A number must now drop below  $.00000001 \times 10^{-38}$  before it is silently replaced by zero.

In the Unnormalized Modes the range of non zero floating point numbers representable in the 7094 is extended downward from  $2^{-129}$  to  $2^{-155}$  in single precision and  $2^{-182}$  in double precision. This allows underflow to take place more gently, and improves the accuracy of certain results. But these benefits are secondary; the primary justification for the Unnormalized Modes is that they ease the task of deciding, in certain cases, whether a result is right or wrong.

For example, consider the following FORTRAN program to compute

$$r = b - \sum_{i=1}^N a_i x_i$$

(In accordance with good computing practice, and because it costs almost nothing extra to do so on our 7094-II, the products of the single-precision numbers  $a_i$  and  $x_i$  are accumulated to double precision before  $r$  is rounded (not truncated) to single precision.)

```

DOUBLE PRECISION D
DIMENSION A(...), X(...)
D = -B
C   ENTER THE UNNORMALIZED MODE.   (14 MICROSEC.)
      CALL FPTUN
DO 1 I=1,N
1   D = A(I)*X(I) + D
C   RESTORE THE STANDARD MODE.     (13 MICROSEC.)
      CALL FPTST
R = 0.0 - RND(D)

```

The last statement rounds D to single precision, changes sign, and adds zero before storing the result in R. If the rounded value of D is a non zero unnormalized number, then the normalization that always follows addition will cause an underflow which, in the Standard Mode, will set R = 0.0 and UNFLOW = .TRUE. . But if RND(D) is a normalized number then adding zero will not change anything. Consequently, R is correct as it stands, despite the possible underflows of intermediate results, with the following exceptions:

- IF OVFLOW OR UNFLOW is .TRUE., R is wrong.
- If severe cancellation has taken place in statement 1, R may be badly contaminated by double-precision truncation errors. This possibility is independent of over/underflow, and is irrelevant if B, A, and X are each uncertain by a unit in their respective last places.
- If R = 0.0 then it may be further contaminated by an error of  $2^{-156}$ , although this is irrelevant if B is non zero and uncertain by a unit in its last place. But if B = 0.0 then all the products A(I)\*X(I) might have underflowed to zero silently.

There are very few applications where any but the first exception is relevant, and that one is caught by the system. The absence of over/underflow tests in the inner loop permits calculations in the normal range to proceed with no noticeable loss of speed.

The Unnormalized Modes may be used in single precision, double precision and complex arithmetic at the cost of 42 microseconds per underflow. These modes would be much more useful on a 7094 but for a quirk in the hardware which forces the "normalized" product of two non zero unnormalized numbers to be zero on certain occasions. The Unnormalized Modes are best suited to those machines, like the Burroughs B 5500, which handle normalized operands without serious anomalies. But, because of the peculiar behaviour of our machine, the Unnormalized Modes are so beset by restrictions (for which see the PRM) that the author and a few of his students are perhaps the only programmers who use them. We find them valuable for computations with matrices, power series, and numerical quadrature.

THE COUNTING MODE. This mode deals with over/underflow in a way which permits programmers to save all the significant digits which are lost by the other modes, and is specially useful for evaluating expressions like

$$q = \prod_{i=1}^N (a_i + b_i) / (c_i + d_i)$$

when  $q$  is likely to be a reasonable number even though its partial products and quotients are afflicted with over/underflow. The execution of

CALL FPTCT(J) ,

where  $J$  is the name of an integer variable, switches .FPTRP in 14 microseconds to the Counting Mode and designates cell  $J$  to act as a leftward extension for the 8-bit characteristics of the AC and MQ registers. Henceforth, over/underflows are counted in  $J$ . Whenever an arithmetic operation overflows its result is divided by  $2^{256}$  and  $J$  is increased by 1. Whenever an arithmetic operation underflows its result is multiplied by  $2^{256}$  and  $J$  is decreased by 1.

For example, the FORTRAN statements

```
CALL FPTCT(J)
J = 0
X = (A+B)*(C+D)*(E/F)/G
```

produce a pair  $(J, X)$  whose values really satisfy

$$(A+B)(C+D)(E/F)/G = 2^{256J} X .$$

In effect, the missing binary digits in  $X$ 's characteristic have been added to  $J$  while  $X$ 's other significant binary digits have remained unchanged.

FORTRAN programmers who use the Counting Mode must be reasonably familiar with the workings of the compiler so that they will not try to evaluate expressions like

$A/(B+C)$     nor     $A*B+C$     nor     $A**B$

in one FORTRAN statement.

The following example shows how the Counting Mode is used to evaluate

$$q = \prod_{i=1}^N (a_i + b_i) / (c_i + d_i)$$

for large  $N$  with no over/underflow tests inside the DO loops, although each over/underflow does cost 35 microseconds.

```

J = 0           Initialize Over/Underflow Counter,
PAB = 1.         Numerator, and
PCD = 1.         Denominator.
CALL FPTCT(J)   Switch to Counting Mode.
DO 1 I=1,N      Compute Denominator using
1   PCD=RND(PCD*RND(C(I)+D(I)))   Rounded Arithmetic.
   IF(PCD .EQ. 0.0) GO TO 3       ... because Denominator vanished.
   J = -J                       Reverse meaning of Counter.
DO 2 I=1,N
2   PAB=RND(PAB*RND(A(I)+B(I)))   Compute Numerator.
   Q = PAB/PCD
   CALL FPTST           Switch back to Standard Mode.
   IF (Q .EQ. 0.0)      J=0       ... because Numerator vanished.
   IF (J) 4, 5, 3
3   ... Q has Overflowed, because J > 0 or Denominator = 0.
4   ... Q has Underflowed, because J < 0 .
5   ... Q is correct as it stands, because J = 0 .

```

Whatever value  $J$  may have, and provided the denominator  $PCD$  is non zero, the stored value  $Q$  is related to the desired value  $q$  by

$$q = 2^{256J} Q .$$

The Counting Mode works for both single and double precision arithmetic, and is indispensable for computing determinants and certain ratios of factorials, but I have not yet figured out how to make a Complex Counting Mode work with comparable elegance on our machine. However, the next example is one where our Counting Mode is useful in a complex arithmetic calculation.

Suppose the complex array  $Z(I)$  is given and we seek  $K$  such that

$$CABS(Z(K)) = \max_{1 \leq I \leq N} CABS(Z(I)) .$$

(Here  $CABS(Z) = |Z|$  in FORTRAN IV.) To avoid the square roots, we may prefer to calculate only squared magnitudes, thereby exploiting the equivalence between the statements

$$(i) \quad |a + ib| > |u + iv|$$

and

$$(ii) \quad a^2 + b^2 > u^2 + v^2 .$$

But the squared magnitudes may over/underflow despite that the magnitudes  $|a + ib|$  and  $|u + iv|$  are well within the machine's range. The following program exploits the equivalent between (ii) above and

$$(iii) \quad (a-u)(a+u) > (v-b)(v+b)$$

and then copes with over/underflows via the Counting Mode.  $N$  is assumed to exceed 1.

```

COMPLEX Z(...), C, W
  DIMENSION ABC(2), UVW(2)
  EQUIVALENCE (C,ABC,A),(B,ABC(2)),(W,UVW,U),(V,UVW(2))
C   This EQUIVALENCE makes c=a+ib and w=u+iv .
CALL FPTCT(J)
K=1                               Initialize current maximum.
C=Z(1)
DO 5 I=2,N
  J=0
  W=Z(I)
  XL = (A-U)*(A+U)
  J= -J
  XR= (V-B)*(V+B)
  IF(XR .EQ. 0. .OR. XL .EQ. 0.) GO TO 3
  IF(J) 2, 3, 1
C   J>0 means |XR| should exceed |XL| , so ignore XL .
1   IF(XR) 5, 5, 4
C   J<0 means |XL| should exceed |XR| , so ignore XR .
2   IF(XL) 4, 5, 5
C   J=0 means XL and XR are directly comparable.
3   IF(XL .GE. XR) GO TO 5
4   K=I                               Update current maximum whenever
    C=W                               W > C .
5   CONTINUE
CALL FPTST

```

Now  $C = Z(K)$  is the largest in magnitude of the values  $Z(I)$  . Some minor refinements can be introduced to reduce the influence of roundoff in critical cases of near equality, but they do not change the relative speed and simplicity exhibited by this program when compared with alternatives. (For more details, see our library program CMAXA in the PRM.)

An attempt was made to extend the idea of FPTCT to cope with integer overflows; i. e. we wanted to allow the FORTRAN programmer to designate



a cell which would act as a leftward extension of the integer accumulator in the same way as J in FPTCT(J) acts as a leftward extension of the floating point accumulator's characteristic. However, this scheme would first have required certain modifications to the 7094 to permit trapping on fixed point overflow, and then the FORTRAN IV compiler would have had to be extensively rewritten. A frustrating feature of the present compiler is that it renders certain integer overflows undetectable! Consequently, FORTRAN programs which manipulate large integers are very much complicated by the need for frequent overflow tests in advance of arithmetic operations. The same complication afflicts ALGOL and any other programming language I know; it is the price we must pay for a lapse in communication among the architects, implementors and users of a programming language.

A similar lapse has frustrated attempts so far to implement the Unnormalized and Counting Modes upon some other machines. The B5500 discards one of the digits in the characteristic of an over/underflowed result, thereby preventing any analysis from determining whether the result over/underflowed by a little or by a lot. The IBM 360 series wantonly destroys everything, including the sign of an overflowed result\*. The CDC 6600 has its own fixed ideas about over/underflow. The tendency of other high-performance machines, like the IBM 360/91, to suffer from imprecise interrupts implies that those machines will have to deal with over/underflow entirely in their hardware. This in turn implies that their treatment of over/underflow will be intolerable unless numerical analysts act soon to lay down reasonable guidelines for machine designers to follow.

IMPROPER DIVISIONS. On a 7094 with divide-check-trap hardware, improper divisions do not turn on the divide-check indicator. Instead they trap to .FPTRP which, in our system, responds as illustrated below.

$1.0/0.0 = 1.7014 \times 10^{38}$  and Overflow occurs.

Any floating point division (single precision, double precision, or complex) of a non zero number by zero is treated as a quotient overflow and sets OVFLOW = .TRUE. . No provision has been made to distinguish such divisions by zero from other quotient overflows (except in the Counting Mode, where a message can be produced) because both events almost always have the

---

\*This sentence was true when it was written; meanwhile IBM has promised to remedy the 360's treatment of over/underflow in a way that may well permit the schemes described here to be copied on the 360's other than 360/91.

W.K. May 1967



same causes and consequences. Besides, the programmer can easily (and should) test directly whether a divisor is zero or not. Each division by zero consumes more than thrice as much time as any other overflow.

1/0 = Kickoff unless otherwise has been requested.

Fixed point integer division by zero is almost certainly a drastic error in a FORTRAN program. In ALGOL the issue might not be so clear.

0.0/0.0 = Kickoff unless otherwise has been requested.

Floating point division of zero by zero is a symptom of a serious flaw in the analysis behind a program.

Unnormalized Division may kick off unless otherwise has been requested.

Floating point division by an unnormalized number causes a trap (unless the quotient produced by the hardware happens to be correct). This is a symptom of certain programming errors like

reference to a variable whose value has not previously been set,

ALOG(3) instead of ALOG(3.0),

a forgotten EQUIVALENCE (A,I) ,

reference to A(13) when DIMENSION A(6) , or

a significant underflow in an Unnormalized Mode.

After the new .FPTRP was installed, failures began to show up in programs which had previously been allowed to proceed silently with a zero quotient for each improper division. A few programmers protested that they liked the old ways better, but they seem to represent a lunatic fringe among programmers as a whole. The author is under the impression that the new .FPTRP's treatment of improper divisions is more widely appreciated than all his other works put together; actually the credit should be shared with R. Jones and J. Bell, who found a way to simulate the divide-check-trap hardware on a 7094 without that equipment. (The equipment is soon to be installed, and with it will come some system simplification.)

However, the most important contribution made by the new .FPTRP is that a programmer who has to cope with a complicated numerical problem can still write whatever program first comes into his mind, just as he did before. And now he will rest assured that, should his algorithm be

frustrated by over/underflow, he will find out what happened and, perhaps, be able to cope with his difficulty by simply re-coding a small part of his program instead of laboriously devising a deeper mathematical analysis of his problem. The new .FPTRP strengthens the programmer's most valuable tool, hindsight.

SIMULATED OVER/UNDERFLOW IN LIBRARY PROGRAMS. The concept of over/underflow is normally associated with the elementary arithmetic operations, but it takes no imagination to extend the concept from simple functions of  $X$  like

$$A+X, \quad A*X, \quad A/X, \quad X**2$$

to more complicated functions like

$$\text{LOG}(X), \quad \text{EXP}(X), \quad \text{COT}(X), \quad \dots$$

In general, it seems reasonable to associate overflow with the following behaviour:

$$\text{as } x \rightarrow x_0 \quad (x_0 \text{ may be } \pm \infty), \quad f(x) \rightarrow \pm \infty.$$

$$\text{e.g. as } x \rightarrow 0^+, \quad \log(x) \rightarrow -\infty;$$

$$\text{as } x \rightarrow +\infty, \quad \exp(x) \rightarrow +\infty.$$

And underflow might just as reasonably be associated with this behaviour:

$$\text{as } x \rightarrow \pm \infty, \quad f(x) \rightarrow 0.$$

e.g.

$$\text{as } x \rightarrow -\infty, \quad \exp(x) \rightarrow 0.$$

But we should not like to associate underflow with the value  $\log(1)=0$ . In other words, underflow occurs only when the value of the function  $f(x)$  is not zero though closer to zero than the underflow threshold.

Here are some examples to illustrate how our functions behave in FORTRAN:

LOG(0.0)	÷ -1.7014 E38	and	OVFLOW	is set
COT(+0.0)	÷ +1.7014		OVFLOW	
EXP(3000.)	÷ 1.7014 E38		OVFLOW	
EXP(-3000.)	= 0.0		UNFLOW	
(+0.0)**(-3.0)	÷ +1.7014 E38		OVFLOW	
0.0**(-3.0)	÷ 1.7014 E38		OVFLOW	
(-100.0)**(-25)	= -0.0		UNFLOW	

The last example is interesting because the IBM program signals overflow during the computation; we avoid overflow by computing  $(1./100)**25$  instead of  $1./(100.**25)$ . The previous two examples should not be confused with

$0**(-3) = \text{Kickoff}$  , code 25 ;

the distinction is consistent with the rules for improper divisions. Finally, no underflows occur when  $\text{LOG}(1.0) = 0.0$  or when  $\text{SINPI}(X) = \sin \pi X$  vanishes for integer values of  $X$ .

I have rewritten several of the elementary function subprograms in the IBLIB library to ensure that their over/underflow behaviour is consonant with the foregoing. When necessary, over/underflow is simulated; this merely means that a transfer to .FPTRP is forced in such a way that the FPT indicator word (cell 0) contains just the information needed for the desired message from .FPTRP. The simplest way to do this in a FORTRAN program is to square a very large or very small number. Of course, .FPTRP must be operating in one of its Standard Modes to allow such simulated over/underflows to produce their intended effects. If the Printing Mode is in use, as it should be while a program is being debugged, then the error-trace points to the function which caused the apparent over/underflow; otherwise the post-execution message may sometimes identify that function. As far as I can see, no vital information is lost by thus failing to discriminate between the simulated over/underflows and the others. The user's view of the library programs becomes less cluttered by their various demands for valid arguments. And the system gains several storage locations vacated by superfluous messages.

However, some programmers claim that one desirable capability has been lost. For example, they would prefer to be able to write

`CALL KIKOPT (9,0)`

in their main program whenever they want references to  $\text{LOG}(X)$  in all their subprograms to cause kickoff when  $X = 0.0$ . My scheme requires that each appearance of  $\text{LOG}(X)$  be preceded by something like

`IF (X .EQ. 0.0) CALL UNCLE (9,18H LOG(X=0.0) ERROR) .`

I think that programs written the second way are easier to read and to debug; but anyone who wants to live dangerously can easily change the library programs to suit himself because their listings are usually amply supplied with comments.

A more penetrating criticism of my scheme is that it denies too many users the valuable education obtained by reading certain IBM diagnostics. For example, increasingly many of our users have too little familiarity with the rate of growth of  $\exp(x)$  to appreciate that  $\exp(88.0297)$  exceeds the overflow threshold. Our university used to include a professor whose first assignment to freshman physics students was to plot a graph of  $\exp(x)$  for  $0 \leq x \leq 10$ . His attitude might well serve as an example for the socially acceptable computer systems of the near future.

The extension of a comprehensive treatment of over/underflow over the entire library of numerical subprograms is an enormous task prodigiously demanding of attention to detail. Here is a simple example of a typical detail. The CABS function computes the absolute value of a complex variable using the formulae

$$\begin{aligned} |a + ib| &= |a| \sqrt{1 + (b/a)^2} && \text{if } |a| > |b| \\ &= |b| \sqrt{1 + (a/b)^2} && \text{if } |b| \geq |a|. \end{aligned}$$

For simplicity assume the former case. Then underflow will occur during the computation of  $1 + (b/a)^2$  whenever  $(b/a)^2$  is non zero but smaller than the underflow threshold. This underflow is irrelevant, so our CABS program suppresses it. Had the program been written in FORTRAN the suppression would have been accomplished by computing  $1 + (b/a)^2$  in the Unnormalized Mode. Similar but more complicated considerations affect the division of one complex number by another.

The task of taming over/underflow in the library is not yet completed; there are several relatively rarely used programs that remain to be revised. Is this project worth its price? Who should say? Our users can no longer offer a qualified opinion because so few of them are now aware of the issues, and even those few hardly ever have trouble dealing with over/underflow nowadays.

**ACKNOWLEDGEMENT.** The author is deeply grateful for the patient assistance rendered by several IBM personnel, both in Toronto and elsewhere, who went out of their way, and sometimes out on a limb, to help with this work. Particular thanks go to J. Leppik, G. Howard and J. Bell for their help with the monitor, the compiler and the revised SAVE pseudo-op. Thanks go as well to colleagues in the Department and in the Institute of Computer Science for their encouragement over several years, and for their help with policy decisions about kick-off and diagnostic procedures.

Some of the work reported here was supported by the National Research Council of Canada.

## ROUNDING ERROR, ILL-CONDITIONING, AND INSTABILITY\*

Ben Noble

Mathematics Research Center, U.S. Army  
University of Wisconsin, Madison, Wisconsin

1. INTRODUCTION. Modern digital computers perform so much arithmetic so rapidly that we can print out only a minute fraction of the results generated within the machine. One of the characteristics of digital computers is that they give a definite answer to everything you ask them to do, whether the answer is right or wrong. The challenge is to write programs in such a way that computations are in some sense self-checking. The more usual situation is that we try as far as possible to incorporate checks, but the printout makes us suspect that something is wrong -- How do we locate the source of the trouble?

The theme of this paper is that it is convenient to subdivide sources of difficulty into three more or less distinct categories. (We go into detail in connection with examples later.)

(a) Existence and uniqueness. It is pointless to look for a unique solution to a problem if there is no solution or an infinity of solutions. If there is an infinity of solutions we may be able to characterize the multiplicity of solutions in a definite way. If there is no solution we may have to look for some approximate solution, for example least-squares or minimax.

(b) Ill-conditioning. Some problems are very sensitive to small changes in the initial data. This is a characteristic of the problem itself, and not of the method used to solve it.

(c) Instability. Some methods for computing the answer to a given problem may be numerically unstable and give nonsensical results, whereas other methods for the same problem may be stable and give accurate results. Instability is a characteristic of the method used to solve the problem, not of the problem itself.

The terms "ill-conditioned" and "unstable" are not always used in exactly these senses in the literature - in particular they are often defined precisely in connection with a particular problem or method. In our usage, the important distinction is that "ill-conditioning" is a property of the problem and "instability" is a property of the method.

If a problem has a well-defined solution that is well-conditioned (i.e., not sensitive to small changes in the given data) we say it is well-posed. Otherwise it is ill-posed. The property of being well-posed or ill-posed is a characteristic of the problem itself, not of the method used to solve it.

---

\* Work performed under Contract No.: DA-31-124-ARO-D-264

One of the reasons why the distinction between existence and uniqueness, ill-conditioning, and instability is convenient is that it corresponds to three stages in the analysis of any given problem:

(i) We cannot compute intelligently until we understand what to look for - a unique solution, a family of solutions, or some kind of an approximate solution. Also a discussion of singular cases will tell us where we should expect difficulties. In general we are likely to be in trouble in situations where we have "nearly" multiple solutions or no solutions at all. If the mathematical theory is inadequate we may be forced into arguments like 'the physical situation has a unique solution so the equations are likely to have a unique solution'. Unfortunately there is not necessarily a one-one correspondence between the physical situation and the mathematical model.

(ii) Once we understand the existence-uniqueness question we can proceed to an analysis of the condition or sensitivity of the problem. Ideally this will tell us when to expect ill-conditioning, and how to recognize it in practice. If a problem is ill-conditioned, the results of a computation are likely to be inaccurate due to rounding errors. Many computers tend to accept ill-conditioning as an act of God. A more satisfactory attitude is to regard it as man-made, and try to develop some ingenious method for avoiding the ill-conditioning, insofar as this is not inherent in the original situation that gave rise to the equations we are trying to solve - for instance, it is sometimes possible to invent a purely mathematical trick as in the least-squares example in §4, or sometimes the physical problem can be reformulated as in the chemical experiment mentioned in §5. To quote J.W. Tukey, "If a job is not worth doing, it is not worth doing well". The accurate solution of an ill-conditioned problem may fall into the class of jobs that are not worth doing, since the results may be meaningless if the initial data are not accurately specified.

(iii) Having understood the problem from a theoretical point of view, we should be in a position to decide which algorithm to use to compute the solution. One of the important properties of an algorithm is that it should be numerically stable. In particular it should not produce spurious solutions and it should not be unduly influenced by rounding errors. An unstable method will be sensitive to rounding errors even though the problem we are trying to solve is itself well-conditioned.

It should not be necessary to remind the reader that, after all this preliminary work has been done, no matter how satisfactory the theory, it is still essential to incorporate checks in programs. When solving differential equations by step-by-step methods, one can perform runs for various step-lengths and check that these give consistent answers. When solving simultaneous equations one can check pivots, and so on. Checks of this type ought to be second nature. Unfortunately many programmers act like the housekeeper who refuses to count up her housekeeping bills more than once - because she always obtains a different answer the second time.

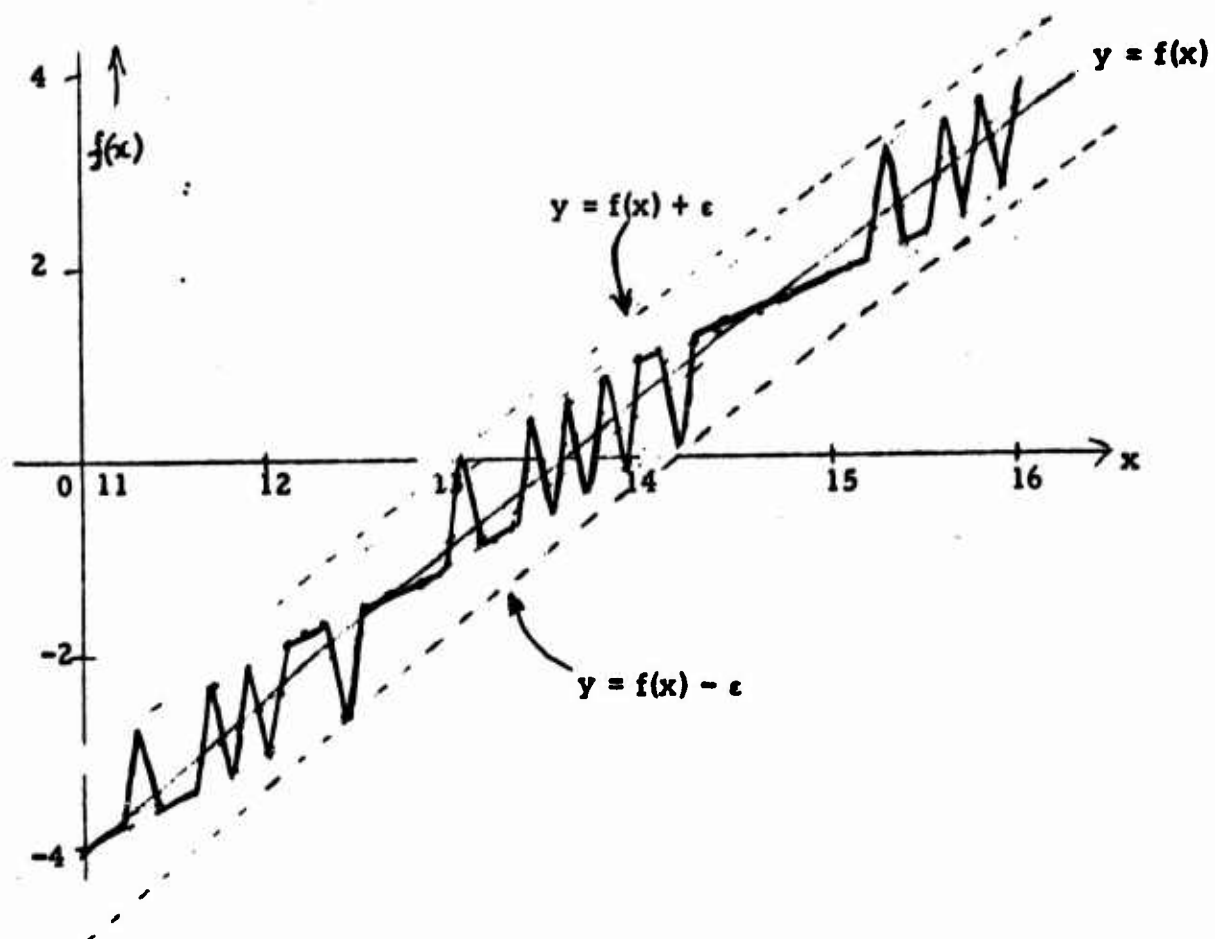


FIGURE 1. Random fluctuations due to rounding when evaluating  $f(x)$ .



2. THE "NOISE-LEVEL" OF A CALCULATION. One of the fundamental limitations inherent in computing is that numbers are specified to a limited degree of accuracy. It will suffice for our purposes to consider floating-point computations with numbers to the base 10, i.e., a number  $x$  is represented in the form  $10^p q$ , where  $p$  is the exponent and  $q$  is the fractional part. The number  $q$  is normalized so that  $0.1 \leq q < 1$ , and  $q$  is specified to a given number of significant figures. The result of a calculation (e.g., an addition or a multiplication) is first normalized and then rounded so that  $q$  always has the same number of digits to the right of the decimal point.

In most cases it is impractical to trace the rounding errors in detail through a calculation. Fortunately the overall effect of rounding errors can be summarized in a simple way. We illustrate by means of a simple example. Suppose that we wish to evaluate

$$f(x) = x - 1000 \{(x + 0.1)^{1/2} - x^{1/2}\}. \quad (1)$$

(We forestall a comment by the expert in numerical analysis, that this particular calculation can be rearranged so that the rounding error is reduced. This remark is irrelevant here since we wish to illustrate what can happen when rounding effects are serious.) On evaluating  $f(x)$  to four significant figures, we have, for example, using the rules for floating-point described in the last paragraph but not floating point notation,

$$\begin{aligned} f(13.40) &= 13.40 - 1000 \{3.674 - 3.661\} = 13.40 - 13.00 = 0.40 \\ f(13.50) &= 13.50 - 1000 \{3.688 - 3.674\} = 13.50 - 14.00 = -0.50 \\ f(13.60) &= 13.60 - 1000 \{3.701 - 3.688\} = 13.60 - 13.00 = 0.60 \end{aligned}$$

These results, together with similarly computed values of  $f(x)$  for  $x$  at intervals of 0.1 from  $x = 11.0$  to  $16.0$  are plotted in Figure 1. (The lines joining the points are of course inserted only to help the eye.) A curve representing the exact value of  $f(x)$ , obtained by using a large number of significant figures in the calculation, is also included. It is seen that the results obtained by using four significant figures fluctuate in a more or less random way about the true  $f(x)$ . The reason why these fluctuations are so large in this case is that there is a serious loss of accuracy because of the subtraction of nearly equal numbers. The more or less random fluctuations of the computed values around the exact curve, as illustrated in Figure 1 is analogous to "noise" in electrical networks.

Mathematically, these results can be stated in a convenient form by saying that if  $f(x)$  is the exact value of a function, and  $f^*(x)$  is the value obtained by evaluating the function on a computer, using a given number of significant figures, then

$$|f^*(x) - f(x)| < \epsilon \quad (2)$$



where the value of  $\epsilon$  cannot be taken smaller than a certain irreducible minimum, depending on the precise way in which the calculations have been performed. Thus in the above example the value of  $f^*(x)$  falls within the two dotted lines, and the minimum permissible value of  $\epsilon$  is, by estimation from the graph, 0.85. This is an empirical estimate of  $\epsilon$ . The quantity  $\epsilon$  is loosely referred to as the noise-level of the calculation, by analogy with fluctuations in electrical networks, for example.

3. POLYNOMIAL EQUATIONS. To illustrate some of the general remarks made in §1, consider the problem of finding the roots of a polynomial equation:

$$a_0 x^n + a_1 x^{n-1} + \dots + a_{n-1} x + a_n = 0,$$

where the coefficients  $a_i$  are real. The mathematical theory for this equation tells us that the following possibilities exist. We assume that  $n$  is an integer greater than or equal to zero.

- (A)  $n = 0, a_0 = 0$ . The equation then reads  $0 = 0$  and any  $z$  is a solution.
- (B)  $n = 0, a_0 \neq 0$ . The equation is then contradictory, since it says that  $a_0 = 0$ . No solution exists.
- (C)  $n > 0, a_n \neq 0$ . The equation has exactly  $n$  roots. Complex roots occur in conjugate pairs.

One of the important things here is that we would normally regard cases (A) and (B) as trivial, but they will give trouble on a computer unless they are allowed for in the computer program. In a general purpose program we must allow for all possibilities, and an existence-uniqueness discussion helps us to understand what these possibilities are.

It is difficult to deal with zero and infinity when using an automatic computer. In place of infinity we have a finite upper limit to the numbers that can be represented within the machine. In place of zero we usually find some small number that has been introduced by rounding errors. We can tell the machine that numbers below a certain limit should be regarded as zero, but we have to be careful about scaling since, in floating point, numbers always carry the same number of significant figures, and a number that is small compared with unity can have a small relative error. Similarly the mere fact that a number is large is no guarantee that it should be regarded as infinite. These difficulties become acute when we try to produce sub-routines that will cope automatically with all eventualities. For a discussion in connection with the solution of quadratic equations (where the problem is already by no means trivial) see [1].

To discuss condition, consider the general equation  $f(z) = 0$ . (This covers transcendental as well as polynomial equations.) Suppose that there is a repeated root of multiplicity  $k$  given by  $z = z_0$ . Suppose that a Taylor series expansion of  $f(z)$  exists near  $z = z_0$ , so that we can write

$$f(z) = \frac{(z-z_0)^k}{k!} f^{(k)}(\xi),$$

where  $\xi$  is a number that tends to  $z_0$  as  $z$  tends to  $z_0$ . Consider the roots of  $f(z) + \epsilon g(z) = 0$ , where  $\epsilon$  is a small parameter. Let  $Z$  be a root of this new equation that tends to  $z_0$  as  $\epsilon$  tends to zero. For small  $\epsilon$ , equation (3) gives, approximately,

$$\frac{(Z - z_0)^k}{k!} f^{(k)}(z_0) + \epsilon g(z_0) \approx 0,$$

or

$$Z \approx z_0 + \left\{ - \frac{k! g(z_0)}{f^{(k)}(z_0)} \right\}^{1/k} \epsilon^{1/k} \quad (3)$$

This expression tells us several things:

(1) It is clear that the multiple roots automatically tend to be ill-conditioned. Thus if  $k = 2$  and  $\epsilon = 10^{-10}$ , we have  $\epsilon^{1/2} = 10^{-5}$ , which is very much larger than  $\epsilon$ .

(2) Consider the special case  $k = 1$ , i.e.,  $z_0$  is a simple root of  $f(z) = 0$ . Then

$$Z \approx z_0 - C\epsilon, \quad C = g(z_0) / f'(z_0). \quad (4)$$

The root will be ill-conditioned if  $g(z_0)/f'(z_0)$  is large. This commonly occurs when there is another root close to  $z_0$ . (If there is a repeated root, then of course  $f'(z_0) = 0$  and  $k > 1$ .) We are tempted to say that roots that are close together will be ill-conditioned. However the situation is more subtle than this. (The following examples are taken from [7] pp. 41-47, where further details can be found.) Consider the polynomial of degree 20 with roots  $z = 1, 2, \dots, 20$ , i.e. the expanded form of

$$(z-1)(z-2) \dots (z-20).$$

If we work out the coefficient of  $\epsilon$  in (4) for the root  $z_0 = 16$  and  $g(z) = z^{19}$  we find

$$|C| \approx 0.24 \cdot 10^{10}.$$

Hence the root  $z_0 = 16$  is very ill-conditioned even though we might think that the roots of the polynomial equation are reasonably well separated. (The small roots of this polynomial equation are well-conditioned.) The roots of  $z^{20} = 1$ , namely the twentieth roots of unity, are equally spaced on a circle of unit radius. We might think that these are very close together, and therefore likely to be ill-conditioned. The reverse is the case. Using (4) we find

$$|C| = 1/20$$

for all the roots, so that all the roots are well-conditioned.

(3) Consider the case of a double root,  $k = 2$ . Equation (3) gives

$$Z = z_0 \pm \left\{ - \frac{2g(z_0)}{f''(z_0)} \right\}^{1/2} \epsilon^{1/2} \quad (5)$$

If the quantity in the parenthesis is negative,  $Z$  may be complex even though  $z_0$  is real. As an example, suppose that we try to solve

$$1.4z^2 - 2.8z + 1.4 = 0,$$

working to two significant figures in the usual formula:

$$\begin{aligned} z &= \{2.8 \pm [2.8^2 - (1.4)^2]^{1/2}\} / 2.8 \\ &= \{2.8 \pm [7.8 - 4(2.0)]^{1/2}\} / 2.8 \\ &= 1.0 \pm 0.161. \end{aligned}$$

The correct answer is that there is a double root  $z = 1$ . (In passing we note that if we know there is a double root, equation (5) suggests that if a numerical procedure produces two roots that are close together and the method is such that the errors are correlated - which is often the case - a much better estimate of the root can be obtained by taking the mean of the two results. In the above example this gives the exact repeated root  $z = 1$ !)

An explicit formula is not usually available for the roots of an equation  $f(z) = 0$ , and most methods of solution will depend on the evaluation of  $f(z)$  for various values of  $z$ . This is true of all iterative methods, for instance - the bisection method, the secant method, straightforward iteration  $z_{r+1} = f(z_r)$ , Newton's method, and so on. The idea of noise-level is useful here. For a simple root, the situation is illustrated graphically in Figure

2(a). Whatever method is used, if the accuracy depends on the accuracy of the evaluation of  $f(z)$ , the best we will be able to do is to say that the root lies somewhere in the range PQ, independent of the method used to find the root. From this point of view, the reason why the situation for a double root is more serious is illustrated in Figure 2(b). Although the noise-level is the same as in Figure 2(a), the range of uncertainty PQ is much greater. If we are unlucky

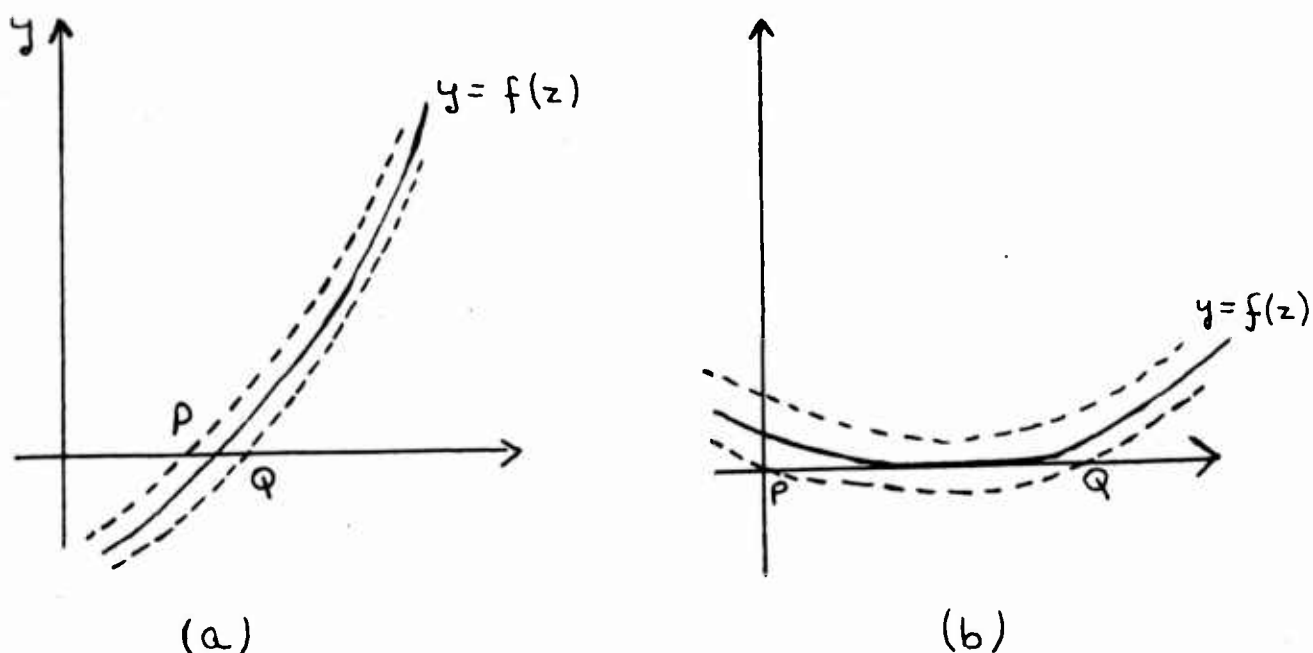


Figure 2. Noise-level and the accuracy of the determination of roots.

and rounding errors cause the machine to produce values of  $f(z)$  that lie above the exact curve in Figure 2(b), the machine may report that there is no root in this region of  $z$ .

We now come to the question of the stability of the algorithm used. Consider solution of the quadratic equation

$$az^2 + bz + c = 0,$$

by means of the usual formula

$$z = \left\{ -b \pm (b^2 - 4ac)^{1/2} \right\} / (2a). \quad (6)$$

Consider

$$z^2 - 100z + 1 = 0.$$

Working to three significant figures in floating point, formula (6) gives

$$z = \{100 \pm (100^2 - 4)^{1/2}\} / 2 = 100 \text{ or } 0.$$

The smaller root has been lost altogether because of cancellation of equal numbers. However if we solve the equation by means of the formulae

$$\left. \begin{aligned} z_1 &= -\text{sgnb}\{b\} + (b^2 - 4ac)^{1/2} / (2a), \\ z_2 &= c/(az_1), \end{aligned} \right\} \quad (7)$$

we obtain  $z_1 = 100$ ,  $z_2 = 0.0100$ . The relative accuracy of  $z_2$  is now good.

In our terminology, (6) is an unstable formula for the numerical solution of a quadratic, whereas (7) is stable. (This type of example has been overworked, but this does not affect its value.)

As a second example of the distinction between stable and unstable, consider the straightforward iteration

$$z_{r+1} = F(z_r).$$

It is well known that any given equation can be arranged in this form in many different ways, some of which give iterations that may converge and some diverge. In our terminology we say that the convergent arrangements are stable, the divergent arrangements are unstable.

To conclude this section, consider solution of the quadratic equation

$$x^2 - 2x - 1.6 = 0$$

by Newton's method for real roots. The iterative formula is

$$x_{r+1} = \frac{x_r^2 - 1.6}{2(x_r - 1)},$$

which gives the following sequence of values, if we start with  $x_0 = 1.4$ :

r	1	2	3	4	5	6
$x_r$	0.45	1.27	0.00	0.80	2.40	1.49

Other starting values give similar results. It is easy to see why the iterates oscillate. The quadratic has complex roots, and graphically (Figure 3) the  $x_{r+1}$  given by Newton's method is the intersection with the x-axis of the tangent to the curve at the point on the curve with abscissa  $x_r$ . Although a cursory examination of the numerical results might cause us to think that the

iteration is unstable, our trouble is in fact due to uniqueness-existence -- we are trying to find a real root that does not exist.

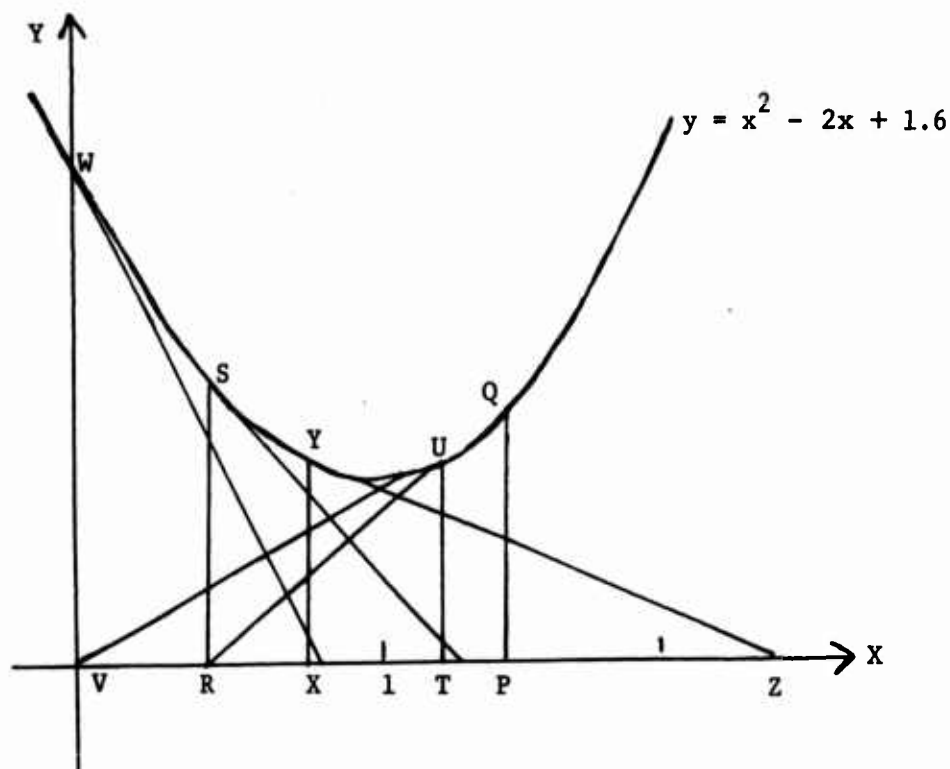


Figure 3. An oscillatory case of the Newton-Raphson procedure. The order of points is PQR...Z.

4. LEAST-SQUARES SOLUTION OF LINEAR EQUATIONS. We begin by briefly summarizing the existence-uniqueness theory for a set of simultaneous linear equations  $Ax = b$ , where  $A$  is  $m \times n$ . There are three possibilities. The equations may have

- (i) No solution.
- (ii) A unique solution.
- (iii) An infinity of solutions.

If an infinity of solutions exist, the general solution can be written in the form

$$x = x_0 + \sum_{i=1}^{n-r} \alpha_i y_i, \quad (8)$$

where  $r$  is the rank of  $A$ , and the  $y_i$  are solutions of the homogeneous equations  $Ay = 0$ . If no solution exists, we often find a solution that minimizes the sum of squares of residuals  $r = b - Ax$ . This least-squares solution can be obtained by solving

$$A^T Ax = A^T b. \quad (9)$$

There are only two possibilities for the solution of these equations -- there may be a unique solution or an infinity of solutions.

The subject of ill-conditioning and linear equations is a long story and this is not the place to go into detail. We content ourselves with the statement that, if  $A$  is square and properly scaled, then a small value for the determinant of  $A$  indicates that the equations  $Ax = b$  are ill-conditioned. (The discerning reader will realize that we are trying to disguise the present unsatisfactory state of the art by not defining what we mean by proper scaling. It is not sufficient to arrange that the largest element in each row and column of  $A$  be of order unity in magnitude.) The result that we wish to make plausible, which is well attested by experience, is that if  $Ax = b$  is ill-conditioned, then the condition of the equations  $A^T Ax = A^T b$  is much worse. This follows when  $A$  is square, if we accept our previous criterion for ill-conditioning since  $\det A^T A = (\det A)^2$ . If  $\det A$  is small, say  $10^{-6}$ , then  $(\det A)^2$  is much smaller still.

The main point we wish to illustrate in this section is that, instead of simply accepting the fact that the condition of (9) may be much worse than the condition of (8), we can do something about it. In the equations  $Ax = b$ , where  $A$  is  $m \times n$  ( $m > n$ ) of rank  $n$ , partition  $A$  and  $b$  in the form

$$A = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix} \quad b = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \quad (10)$$

where  $A_1$  is a nonsingular matrix of order  $n$ , the choice of which will be discussed later, and  $b_1$  is  $n \times 1$ . Since  $A_1$  is nonsingular, the last  $m - n$  rows of  $A$  can be expressed as linear combinations of its first  $n$  rows. This means that we can find a matrix  $P$  such that

$$A_2 = PA_1, \quad (11)$$

i.e.

$$A = \begin{bmatrix} I \\ P \end{bmatrix} A_1. \quad (12)$$

On inserting this expression for  $A$ , together with  $b$  from (10), in the least squares equations

$$A^T A x = A^T b,$$

we find

$$A_1^T \begin{bmatrix} I, P^T \end{bmatrix} \begin{bmatrix} I \\ P \end{bmatrix} A_1 x = A_1^T \begin{bmatrix} I, P^T \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \quad (13)$$

i.e.

$$A_1^T [I + P^T P] A_1 x = A_1^T [b_1 + P^T b_2]. \quad (14)$$

Since  $A_1$  is nonsingular, we can multiply through by  $(A_1^T)^{-1}$ :

$$[I + P^T P] A_1 x = b_1 + P^T b_2. \quad (15)$$

This is the required form of the least-squares equations. We claim that if the set of equations

$$A^T A x = A^T b$$

is very badly conditioned, the condition of the set (15) will be much better, provided  $A_1$  is chosen properly. Before discussing how to choose  $A_1$  we remark that (15) can be rearranged in the form

$$A_1 x = b_1 + [I + P^T P]^{-1} P^T [b_2 - P b_1]. \quad (16)$$

If the last  $m-n$  equations in  $Ax = b$  are simply linear combinations of the first  $n$  equations this means that if  $P$  is defined as in (11) then we must also have  $b_2 = P b_1$ . This means that the second term on the right of (16) vanishes, and we find the least-squares solution by simply solving  $A_1 x = b_1$ , as we should expect. If the equations  $Ax = b$  arise in a physical situation then we should expect that the last  $m-n$  equations would be nearly equal to linear combinations



of the first  $n$ , i.e.,  $b_2$  would be nearly equal to  $Pb_1$  and the last term in (16) will be a small correction to  $b_1$ .

The step that improved the condition of  $A^T Ax = A^T b$  was the multiplication of (14) by  $(A_1^T)^{-1}$  to give (15). Since  $I + P^T P$  is positive definite, with determinant greater than unity, the condition of (15) is determined essentially by the condition of  $A_1$ . How do we choose  $A_1$ ?

The result of 150 years work on the numerical solution of simultaneous linear equations is that Gaussian elimination is still the best general purpose method if precautions are taken to choose the pivots correctly. In the terminology of §1, Gaussian elimination is an unstable computing procedure when rounding errors are present unless the pivots are chosen in a suitable way. The usual rule is to use either partial or complete pivoting. We illustrate by an example. Suppose that we are working in floating point to two significant figures. Consider the equations

$$\begin{aligned} x_1 - x_2 &= 0 \\ 10^{-2}x_1 + x_2 &= 1, \end{aligned} \tag{17}$$

which have the exact solution  $x_1 = x_2 = 100/101$ . To solve these numerically we can use the first equation to eliminate  $x_1$  from the second. In more technical language, we use the coefficient of  $x_1$  in the first equation as pivot. If we multiply the first equation by  $10^{-2}$  and subtract from the second, we obtain  $-1.01x_2 = -1$ . However we are working to two significant figures, so 1.01 is rounded to 1.0, and this equation gives  $\hat{x}_2 = 1$ , where a cap is used to denote "computed value." Back-substitution in the first equation gives  $\hat{x}_1 = 1$ , and we have obtained a reasonable approximate solution of the equations. Suppose however that we pivot on the coefficient of  $x_1$  in the second equation in (17). We multiply the second equation by  $10^2$  and subtract from the first. As before on rounding this gives the computed result  $\hat{x}_2 = 1$ , but back-substitution, now in the second equation, gives  $\hat{x}_1 = 0$ . In this case the computed solution is no longer a reasonable approximation to the exact solution. The only difference has been in the choice of pivots, and this illustrates that the choice of pivots is important. The reader may have gained the impression that the reason why the first solution was satisfactory, whereas the second was not, is connected with the fact that the pivot used in the first case (1) is greater than the pivot used in the second ( $10^{-2}$ ). This is the assumption behind complete and partial pivoting. Partial pivoting tells us to pick the largest coefficient of the variable we propose to eliminate, for instance. It is easy to rescale the set of equations (17) so that partial or complete pivoting is unsatisfactory. The nub of the matter is that we are working in

floating point so that it is only relative error that is important, whereas pivotal strategies usually depend on criteria involving absolute magnitudes. Suppose that we rescale (17) by multiplying the first equation by  $10^{-2}$ , the second by  $10^2$ , and set  $x_1 = 10^2 z_1$ ,  $x_2 = 10^{-2} z_2$ . The equations become

$$\begin{aligned} z_1 - 10^{-4} z_2 &= 0 \\ 10^2 z_2 + z_2 &= 10^2. \end{aligned} \tag{18}$$

If we pivot on the "large" coefficient  $10^2$  we find, working to two significant figures in floating point,  $\hat{z}_2 = 10^2$ ,  $\hat{z}_1 = 0$ , which gives  $\hat{x}_2 = 1$ ,  $\hat{x}_1 = 0$ , i.e., the same unsatisfactory solution found by pivoting on the corresponding coefficient in (17). (It is easy to see that, for a given choice of pivots, rescaling by powers of 10 will not affect the relative rounding errors.)

However if we pivot on the "small" coefficient  $10^{-4}$  in (18) we obtain the satisfactory approximate solution  $\hat{z}_1 = 10^{-2}$ ,  $\hat{z}_2 = 10^2$ ,  $\hat{x}_1 = \hat{x}_2 = 1$ .

The moral of this discussion is that success of partial or complete pivoting depends on proper scaling. Various arguments indicate that it is reasonable to scale so as to minimize the condition number  $\|A\| \|A^{-1}\|$ .

This can be done for the infinity-norm, for instance, by arranging that the absolute row sums of  $A$  are the same, and those of  $A^{-1}$  are the same (F.L. Bauer). For further discussion and references to the work of Wilkinson, Bauer, and others, see [2].

The question of pivotal strategy is relevant to the choice of  $A_1$  in the decomposition (10) used in the method suggested above for avoiding ill-conditioned least-squares equations. If we use either partial or complete pivoting to reduce  $A$  to row-echelon form this will single out  $n$  rows of  $A$ . We choose  $A_1$  to consist of these  $n$  rows. The value of  $\det A$  is the product of the pivots.<sup>1</sup> By using partial or complete pivoting we are trying to choose an  $A_1$  whose determinant is as large as possible. This should be the submatrix of order  $n$  from  $A$  that is as well-conditioned as possible.

5. ORDINARY DIFFERENTIAL EQUATIONS. A great deal is known about the existence and uniqueness of solutions of ordinary differential equations. Rather than go into detail we simply quote [3], pp. 15, 112, 347, for typical theorems that are likely to be useful when computing. We also remind the reader of some simple examples where the conditions for existence or uniqueness of solutions of  $y' = f(x,y)$  are not satisfied. The equation

$$y' = 1 + y^2$$

has the solution  $y = \tan(x+c)$ , where  $c$  is an arbitrary constant, and this solution does not exist when  $x = (n + 1/2)\pi - c$ .

$$y' = |y|^{1+\epsilon}, \quad y(0) = 1, \quad \epsilon > 0,$$

has the solution  $y(x) = (1 - \epsilon x)^{-1/\epsilon}$  which ceases to exist at  $x = 1/\epsilon$ .

If

$$y' = |y|^{1-\epsilon}, \quad y(0) = 0, \quad \epsilon > 0,$$

we have an infinity of solutions:

$$y(x) = 0, \quad 0 \leq x \leq c, \quad y(x) = [\epsilon(x-c)]^{1/\epsilon}, \quad \epsilon > 0,$$

for arbitrary  $c > 0$ .

Existence and uniqueness questions arise when resonance occurs in a physical system. A simple example occurs in connection with

$$\frac{d^2 y}{dx^2} + \lambda^2 y = f(x), \quad 0 \leq x \leq \pi, \quad y(0) = y(\pi) = 0.$$

If  $\lambda = n\pi$ , for integral  $n$ , the homogeneous equation has the solution  $y = \sin nx$ . The situation then is: Let

$$k = \int_0^\pi f(x) \sin nx \, dx.$$

There are two possibilities:

- (i) If  $k = 0$  the equation has an infinity of solutions.
- (ii) If  $k \neq 0$  the equation has no solutions.

We next make some remarks about ill-conditioning. A typical situation is that small changes in the initial conditions, in an initial-value problem, produce large changes in the answer. We consider an example where this is caused by the presence of exponential solutions. Consider

$$y' = \alpha y + (\beta - \alpha) e^{\beta x}, \quad y = y_0 \text{ at } x = 0. \quad (19)$$

The general solution is

$$y = (y_0 - 1)e^{\alpha x} + e^{\beta x}. \quad (20)$$

If  $y_0 = 1$  then  $y = e^{\beta x}$ , and if  $y_0 = 1 + \epsilon$  we have  $y = \epsilon e^{\alpha x} + e^{\beta x}$ . If  $\alpha > \beta$  we will have  $e^{\alpha x} \gg e^{\beta x}$  for large enough  $x$ , and the first term on the right of (20) will dominate the second for large  $x$ , no matter how small  $\epsilon$  is. As

an example, if  $\alpha = 10$ ,  $\beta = -1$ , we find

$$\begin{array}{lll} y_0 = 1 & y = e^{-x} & y(1) \approx 0.37 \\ y_0 = 1.0001, & y = e^{-x} + 0.0001e^{10x}, & y(1) \approx 2.57. \end{array}$$

A change of 1 in  $10^{-4}$  in the initial condition produces a change of 7 to 1 in the solution at  $x = 1$ , and the difference is even more catastrophic for larger  $x$ . The problem is obviously ill-conditioned. We have already said that if a problem is ill-conditioned we should try to reformulate it in some way so that the ill-conditioning is removed. It is possible to do this in the present case if we know that the solution tends to zero as  $x$  tends to infinity. We can use this to replace the initial condition. Thus

$$y' = 10y + e^{-x}, \quad y(0) = 1, \quad (21)$$

is an ill-conditioned problem, but

$$y' = 10y + e^{-x}, \quad y \rightarrow 0 \text{ as } x \rightarrow \infty, \quad (22)$$

is well-conditioned. This last problem can be solved satisfactorily by integrating back from large  $x$  towards the origin. The two formulations (21) and (22) are mathematically equivalent.

We next turn our attention to computational difficulties, not present in the original differential equation, but introduced by the difference scheme used to solve the equation numerically. In this connection the word "instability" is used in a technical sense, for details of which we refer the reader to the literature [3], [4]. Roughly speaking, a difference scheme is said to be unstable if it introduces spurious solutions that are not present in the original problem, and these dominate the solution we want to find, if we are integrating over a fixed range of  $x$ , and we let the step-size in the difference scheme tend to zero. This type of difficulty is now well understood and will not be considered further here.

A common source of trouble is illustrated by the example

$$y'' + 101 y' + 100 y = 0, \quad y = 0, \quad y' = 99, \text{ at } x = 0. \quad (23)$$

The exact solution is

$$y = e^{-x} - e^{-100x}.$$

Two common difficulties when trying to compute this solution are:

(i) The programmer realizes that the solution  $e^{-100x}$  is negligible whenever  $x$  is greater than about 0.05, leaving only  $e^{-x}$ . He therefore

adjusts the step-length  $h$  to be suitable for this part of the solution taking, perhaps,  $h = 0.02$ . However the step-length that must be used in the standard-type difference formula is still controlled by the  $e^{-100x}$  term, even though this is negligible in the actual solution.

(ii) The programmer realizes that the  $e^{-100x}$  term controls the step-length, so he takes  $h$  to be, say,  $0.0001$ . Then he complains that the computation takes an interminable length of time on a digital computer.

This is a typical boundary-layer problem. The highest order derivative is important only over a small part of the range. In this example, one answer would be to take short steps from  $x = 0$  to  $x = x_1$ , say, where  $x_1$  is chosen so that the contribution from the  $e^{-100x}$  term is negligible. Suppose that we find that  $y = y_1$  at  $x = x_1$ . If we did not know the exact situation, we might then simply drop the  $y''$  term in (23), having checked computationally that it is small compared with the other two terms, and solve:

$$101 y' + 100 y = 0, \quad y = y_1 \quad \text{at } x = x_1,$$

which would give a reasonable approximation to the correct solution quite quickly. The calculation can be checked by varying the point  $x_1$  at which the changeover occurs.

Boundary layer phenomena often occur in connection with boundary value problems. A typical example is:

$$\epsilon y'' + y = 0, \quad \epsilon > 0, \quad y(0) = 0, \quad y(1) = 1.$$

For small  $\epsilon$  the solution is almost zero except near  $x = 1$ . A more unusual example is:

$$\epsilon (y'')^2 + xy' - y = 0, \quad \epsilon > 0, \quad y(1) = y(-1) = 1. \quad (24)$$

Neglecting the second derivative we have  $xy' - y = 0$ , with solution  $y = Cx$ , where  $C$  is an arbitrary constant. The solution must be symmetrical about the  $y$ -axis, and the first possibility that suggests itself is that  $y = 0$  over most of the range, with boundary layers at the end-points. However this would mean that near  $x = 1$  the value of  $y'$  would be large and positive, which is not possible since (24) would then imply that  $\epsilon (y'')^2$  is negative which is impossible. It turns out that the solution is approximately  $y = x$  in most of  $0 < x \leq 1$ ,  $y = -x$  in most of  $-1 \leq x < 0$ , and these solutions are joined by a "corner layer" near  $x = 0$ . I am indebted to Carl Pearson for this example. He has also made the sensible remark that in many of these problems an effective computational procedure if  $\epsilon = 10^{-12}$ , say, is to compute a series of solutions with  $\epsilon = 10^{-2}, 10^{-4}, \dots$ , in turn. The computations with the larger  $\epsilon$  will be less difficult, and they will provide successive guides that tell us where boundary layers are developing, and how sharp they are.

To conclude this section we draw the reader's attention to a quite different type of example considered in detail in [6], and in a simplified

form in [5], Chap. 12. A chemical reaction involving three species with concentrations  $a_i$ ,  $i = 1, 2, 3$ , is governed by a system of three linear differential equations in three unknowns:

$$da_1/dt = -(k_{21} + k_{31})a_1 + k_{12}a_2 + k_{13}a_3 \quad (25)$$

with two similar equations for  $da_2/dt$ ,  $da_3/dt$ . The solution of these equations is known to be of the form

$$a_1 = c_{11} + c_{12}e^{-\mu t} + c_{13}e^{-\nu t}, \quad (26)$$

with similar expressions for  $a_2$  and  $a_3$ . The concentrations  $a_i$  can be measured experimentally for various values of the time. It is required to find the rate constants  $k_{ij}$  that appear in (25). The most obvious procedure is to use curve fitting with exponentials to deduce from (26) the values of  $\mu$ ,  $\nu$ , and the  $c_{ij}$ . Then deduce the  $k_{ij}$  from the fact that (26) is the solution of (25).

Unfortunately fitting of exponentials is an ill-conditioned procedure. It turns out that if we perform a detailed analysis of the relation between (25) and (26), a procedure can be devised that will enable the experimentalist to design his experiment in such a way that he can find initial concentrations such that either  $c_{12}$  or  $c_{13}$  is zero in (26). It is then possible to deduce the rate constants by a well-conditioned procedure. Details can be found in the references. From our point of view the moral is again that if one method for performing a calculation is ill-conditioned we should look for an equivalent well-conditioned procedure.

6. CONCLUDING REMARKS. We have illustrated the existence-uniqueness, ill-conditioning, and instability classification of difficulties by discussing various aspects of three types of problem-polynomial equations, least-squares solution of linear equations, and ordinary differential equations. We could equally well have illustrated the classification by discussing other standard problems in numerical analysis - eigenvalues - eigenvectors, approximation theory, partial differential equations, integration, integral equations.

In the lecture from which this paper originated, the three-way classification was also characterized as follows:

(1) Ignorance -- If we try to find a real root of a polynomial when all the roots are complex, this is simply ignorance of the existence-uniqueness situation.

(2) Cussedness -- Ill-conditioned problems are inherently troublesome - the difficulty stems from the nature of the problem, and often there is little we can do about the problem as it stands. The best remedy is to circumvent our difficulties.

(3) Stupidity -- Instability troubles are usually due to the fact that we are not clever enough to choose the correct computational method. Perhaps this is rather a harsh term to apply to situations where foolproof computational methods are not yet known - such as the choice of pivots in Gaussian elimination.

Briefly, if a problem gives trouble, we must first decide whether we are simply ignorant of the existence-uniqueness theory. If we are sure that we are looking for the correct type of solution, we must decide whether the problem itself is cursed (in which case it is probably best to try to reformulate it) or whether we have simply been stupid in our choice of method. My own experience is that this procedure has been useful when trying to track down sources of trouble -- But when all is said and done, and the source of difficulty has been located the most appropriate comment is often 1 Corinthians, Chap. I, v.27 - "God hath chosen the foolish things of the world to confound the wise."

#### REFERENCES

- [1] G.E. Forsythe, What is a satisfactory quadratic equation solver?, Tech. Rep. CS 74, Computer Sciences Dept., Stanford Univ. Aug 7, 1967.
- [2] G.E. Forsythe and C.B. Moler, Computer Solution of Linear Algebraic Systems, Prentice-Hall (1967).
- [3] P. Henrici, Discrete Variable Methods in Ordinary Differential Equations, Wiley (1962).
- [4] E. Isaacson and H.B. Keller, Analysis of Numerical Methods, Wiley (1966).
- [5] B. Noble, Applications of Undergraduate Mathematics in Engineering, Macmillan (1967).
- [6] J. Wei and C.D. Prater, The structure and analysis of complex reaction systems, Advances in Catalysis, Vol. 13, Academic Press (1962), pp. 203-392.
- [7] J. Wilkinson, Rounding Errors in Algebraic Processes, Prentice Hall (1963).



## ATTENDANCE LIST

Gerald D. Adams, Edgewood Arsenal  
Roy E. Barnette, ACIC, St. Louis, Missouri  
Mary A. Biagioli, U. S. Army Aviation Materiel Command,  
St. Louis, Missouri  
Howard M. Bloom, Harry Diamond Laboratories  
Colin Cryer, Mathematics Research Center  
James R. Davis, Sierra Vista, Arizona  
Don Define, ACIC, St. Louis, Missouri  
F. G. Dressel, Army Research Office-Durham  
R. M. Dunn, U. S. Army Electronics Command, Ft. Monmouth  
Edmund A. Early, Army Map Service  
S. H. Eisman, Frankford Arsenal  
Dennis A. Flaherty, Aberdeen Proving Ground  
James E. Frese, Dugway Proving Ground  
Charles A. Funn, U. S. Army Map Service  
John H. Giese, BRL, Aberdeen Proving Ground  
Joseph R. Gillis, White Sands Missile Range  
William D. Googe, Army Map Service  
B. C. Gray, Fort Detrick  
Julia H. Gray, Mathematics Research Center  
C. Maxson Greenland, Edgewood Arsenal  
Donald Greenspan, Mathematics Research Center  
James F. Jacobs, Fort Detrick  
W. M. Kahan, University of Toronto  
Donald F. Kennedy, University of Georgia, COSMIC  
S. C. Kleene, Acting Director, Mathematics Research Center  
Leon Leskowitz, U. S. Army Electronics Command, Ft. Monmouth  
Bert Levy, Harry Diamond Laboratories  
Ralph London, University of Wisconsin  
Frank R. Loscalzo, University of Wisconsin  
Roger A. MacGowan, U. S. Army Missile Command, Redstone Arsenal  
David S. Marsh, Harry Diamond Laboratories  
C. Masaitis, BRL, Aberdeen Proving Ground  
Forrest McMains, Picatinny Arsenal  
John F. Mescall, U. S. Army Materials Res Agency, Watertown, Mass.  
Ramon Moore, University of Wisconsin  
Mervin E. Muller, University of Wisconsin  
Leonard F. Nichols, Picatinny Arsenal  
Ben Noble, MRC and University of Wisconsin  
Stuart Olson, Rock Island Arsenal  
Victor Pereyra, Mathematics Research Center  
Robert G. Polk, U. S. Army Missile Command, Redstone Arsenal  
Mary Rita Powers, U. S. Navy Underwater Sound Lab., Ft. Trumbull

L. B. Rall, Assistant Director, Mathematics Research Center  
Allen Reiter, Lockheed Research Laboratory  
W. E. Sanburn, GIMRADA, Fort Belvoir  
James F. Smith, U. S. Army Engineer Waterways Experiment Station  
B. T. Smith, University of Toronto  
William B. Stelwagon, US Naval Ordnance Test Station, China Lake  
T. D. Streeter, Rock Island Arsenal  
Server Tasdemiroglu, ATAC, Warren, Michigan  
Joseph S. Tyler, Edgewood Arsenal  
Ronald P. Uhlig, U. S. Army Management Systems Support Agency,  
The Pentagon  
James W. Walker, Army Map Service  
Randall K. Walters, Atmospheric Sciences Office, White Sands Missile  
Range, New Mexico  
Richard D. Whittaker, U. S. Navy Underwater Sound Lab, Ft. Trumbull  
Joseph E. Wilson, Rock Island Arsenal  
J. M. Yohe, Mathematics Research Center  
D. E. Zilmer, U. S. Naval Ordnance Test Station, China Lake

Unclassified  
Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author) U.S. Army Research Office-Durham Box CM, Duke Station Durham, North Carolina 27706		2a. REPORT SECURITY CLASSIFICATION Unclassified	
		2b. GROUP NA	
3. REPORT TITLE Proceedings of the 1967 Army Numerical Analysis Conference			
4. DESCRIPTIVE NOTES (Type of report and inclusive dates) Interim Technical Report			
5. AUTHOR(S) (First name, middle initial, last name)			
6. REPORT DATE November 1967		7a. TOTAL NO. OF PAGES 230	7b. NO. OF REFS
8a. CONTRACT OR GRANT NO.		9a. ORIGINATOR'S REPORT NUMBER(S) ARO-D Report 67-3	
b. PROJECT NO.			
c.		9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)	
d.			
10. DISTRIBUTION STATEMENT This document is subject to special export controls and each transmittal to foreign nationals may be made only with prior approval of the U.S. Army Research Office-Durham			
11. SUPPLEMENTARY NOTES None		12. SPONSORING MILITARY ACTIVITY Army Mathematics Steering Committee on behalf of the Office of the Chief of Research & Development	
13. ABSTRACT This is the technical report resulting from the 1967 Army Numerical Analysis Conference. It contains 12 papers, which treat various computer and computational problems of interest to the scientific world.			
14. Key Words  computer programs machine language programming nonlinear systems FORMAC random event generator internal arithmetic homeostatic organizations automatically sequenced events digital-analogue simulation shell computer program numerical solution of polynomial equations			

DD FORM 1473

REPLACES DD FORM 1473, 1 JAN 64, WHICH IS OBSOLETE FOR ARMY USE.

Unclassified  
Security Classification

AD825963



DEPARTMENT OF THE ARMY  
U. S. ARMY RESEARCH OFFICE-DURHAM  
BOX CM, DUKE STATION  
DURHAM, NORTH CAROLINA 27706

IN REPLY REFER TO:  
CRDARD-IPL

3 April 1968

Defense Documentation Center  
Cameron Station  
Alexandria, Va. 22314

Gentlemen:

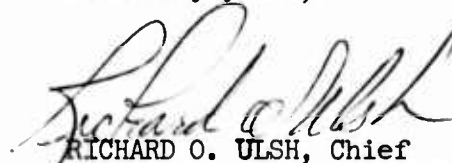
Previously we sent you twenty copies of ARO-D Report 67-3,  
Proceedings of the 1967 Army Numerical Analysis Conference,  
Sponsored by the Army Mathematics Steering Committee on  
Behalf of The Office of the Chief of Research and Development.

Page 187-a was inadvertently not printed with the report, so  
I am inclosing twenty copies of that page so your copies of the  
report will be complete.

We do not have the AD number of this report as our DDC Form 50  
has not yet been returned with that information. I am inclosing  
another copy of Form 50 with these pages.

Thank you for your cooperation.

Sincerely yours,

  
RICHARD O. ULSH, Chief  
Information Processing Office

incls  
as

AD-825-963

monitor to forget the last reference to KICKED whenever overlay occurs. We take no pride in this expedient.

Any programmer who is aware of these two limitations can easily code around them. Simple suggestions are contained in the PRM. Indeed, the limitations are so easy to circumvent that programmers sometimes forget to do so, and for this reason we have included a warning message like the one in the following example:

0.0/0.0 ERROR AT 14506  
EXECUTION TERMINATED.

ERROR-TRACE WITH CALLS IN REVERSE ORDER CODE 25

CALL IS IN DECK NAMED	AT IFN OR LINE NO.	ABSOLUTE LOCATION
SUB2	17+	14513
SUB1	25	07762
MAIN	2	05413

EXECUTING IFN/LINE NO. 2 OF 'SUB1' AFTER PROGRAM WAS KICKED OFF. FROM NOW ON IN 'SUB1', THE VALUE OF A SUBSCRIPTED VARIABLE WITH VARIABLE SUBSCRIPT, OR THE EXECUTION OF A COMPUTED 'GO TO' OR 'DO' STATEMENT WITH VARIABLE PARAMETER, MAY BE INCORRECT UNLESS THE RELEVANT INDEX IS RESET. SEE THE PROGRAMMERS' REFERENCE MANUAL.

This message is more formidable than necessary. It would be unnecessary altogether if the IF(KICKED(OFF)) statement were implemented in a language, like ALGOL, with a block structure. Then kick-off within a block would cause control to be transferred to the last KICKED reference, if any, executed in the same block but not in a deeper sub-block.

One other complication would arise were the IF(KICKED(OFF)) statement to be implemented within a compiler which contained a MONITOR statement. Such a statement is exemplified by

MONITOR X, Y(\*), Z(\*, 3), PROG, n

which would cause output of the following kind to be generated:

Whenever the variable X is changed, write out its new value;

X = 14.271434 .